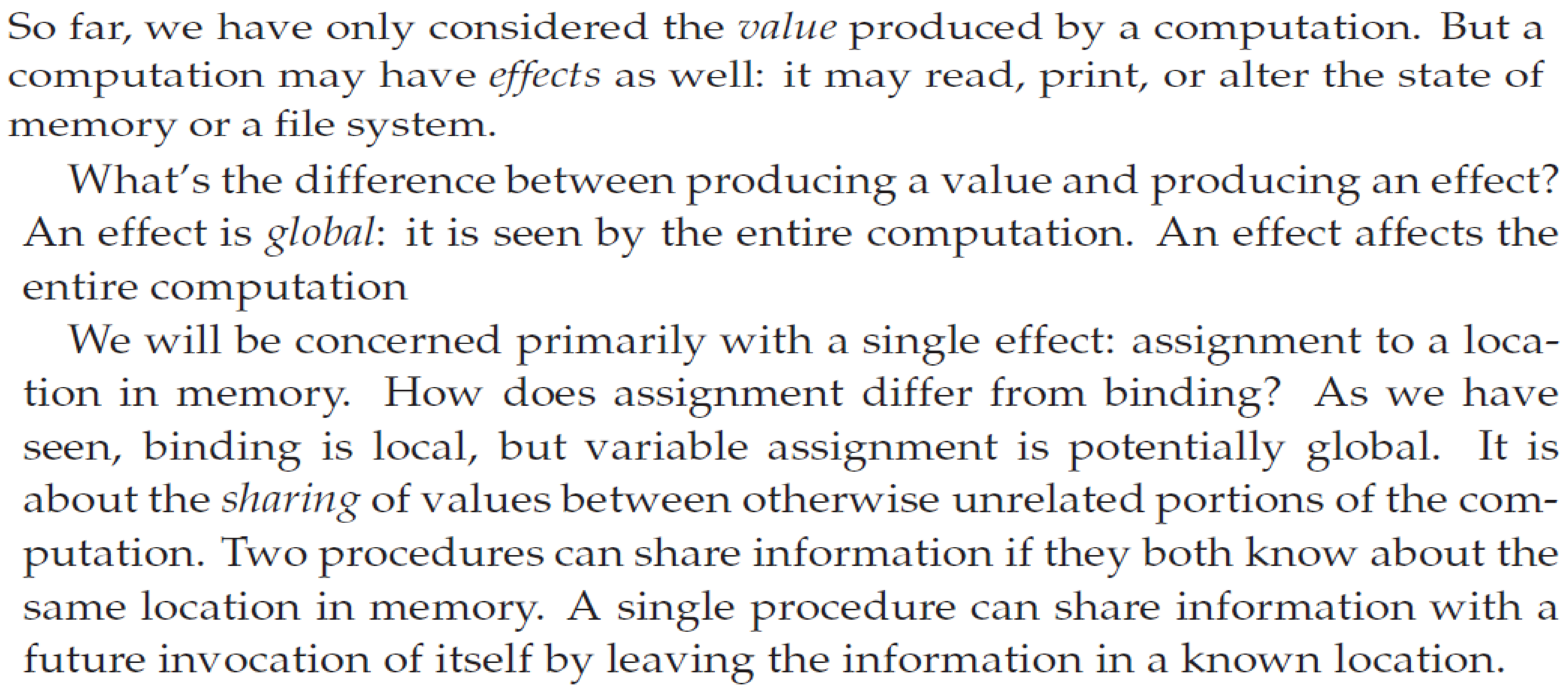
States

Essentials of Programming Languages (Chapter 4)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Computational Effects

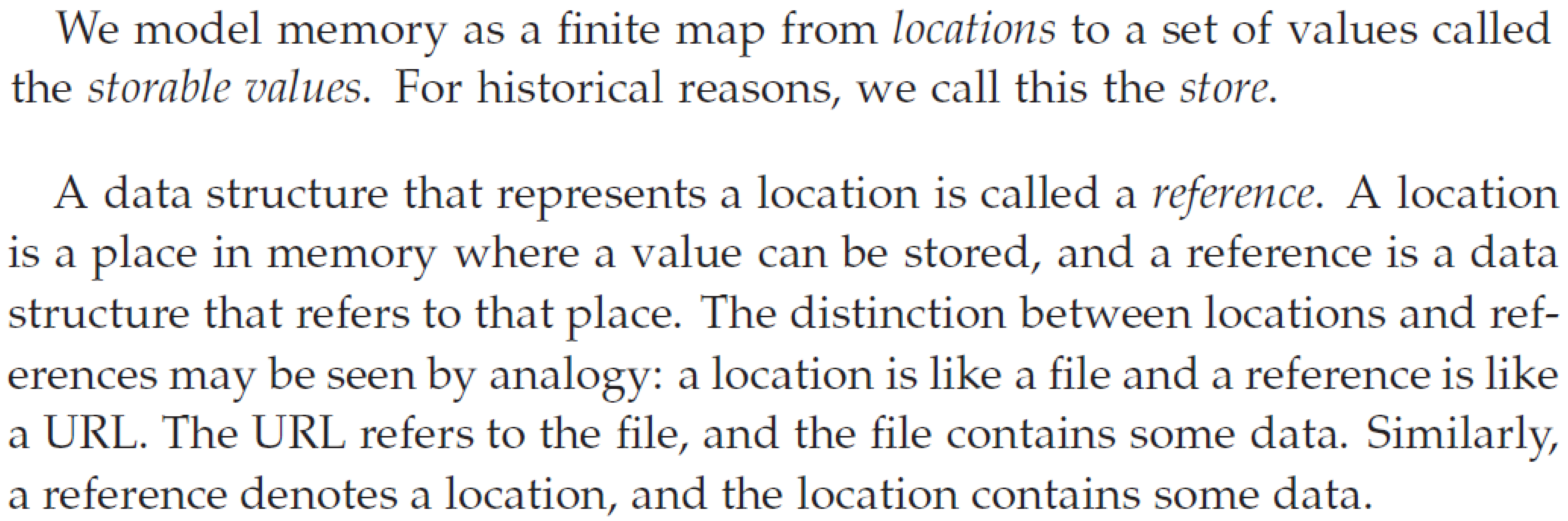


So far, we have only considered the *value* produced by a computation. But a computation may have *effects* as well: it may read, print, or alter the state of memory or a file system.

What's the difference between producing a value and producing an effect? An effect is *global*: it is seen by the entire computation. An effect affects the entire computation

We will be concerned primarily with a single effect: assignment to a location in memory. How does assignment differ from binding? As we have seen, binding is local, but variable assignment is potentially global. It is about the *sharing* of values between otherwise unrelated portions of the computation. Two procedures can share information if they both know about the same location in memory. A single procedure can share information with a future invocation of itself by leaving the information in a known location.

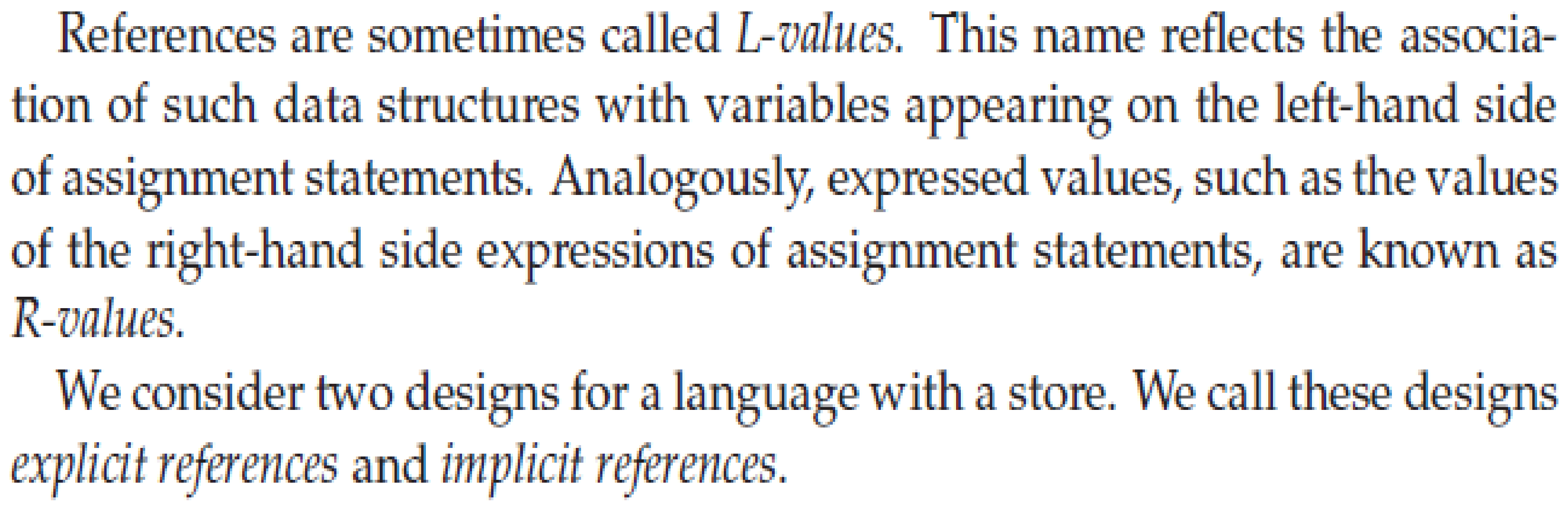
Memory



We model memory as a finite map from *locations* to a set of values called the *storable values*. For historical reasons, we call this the *store*.

A data structure that represents a location is called a *reference*. A location is a place in memory where a value can be stored, and a reference is a data structure that refers to that place. The distinction between locations and references may be seen by analogy: a location is like a file and a reference is like a URL. The URL refers to the file, and the file contains some data. Similarly, a reference denotes a location, and the location contains some data.

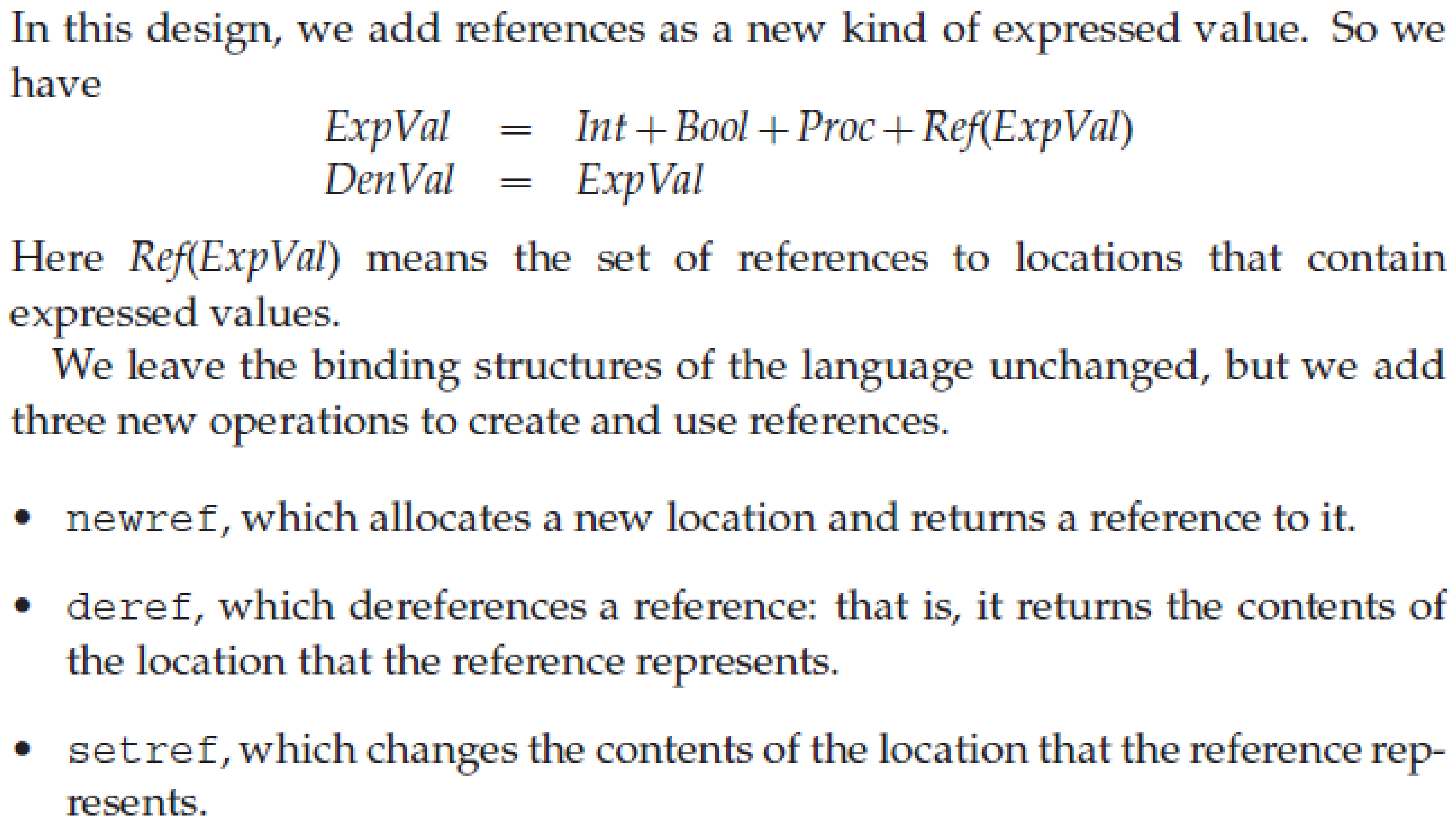
…



References are sometimes called *L-values*. This name reflects the association of such data structures with variables appearing on the left-hand side of assignment statements. Analogously, expressed values, such as the values of the right-hand side expressions of assignment statements, are known as *R-values*.

We consider two designs for a language with a store. We call these designs *explicit references* and *implicit references*.

EXPLICIT-REFS: A Language with Explicit References



In this design, we add references as a new kind of expressed value. So we have

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} + \mathit{Ref}(\mathit{ExpVal}) \\ \mathit{DenVal} &= \mathit{ExpVal} \end{aligned}$$

Here $\mathit{Ref}(\mathit{ExpVal})$ means the set of references to locations that contain expressed values.

We leave the binding structures of the language unchanged, but we add three new operations to create and use references.

- `newref`, which allocates a new location and returns a reference to it.
- `deref`, which dereferences a reference: that is, it returns the contents of the location that the reference represents.
- `setref`, which changes the contents of the location that the reference represents.

EXPLICIT-REFS: Example1

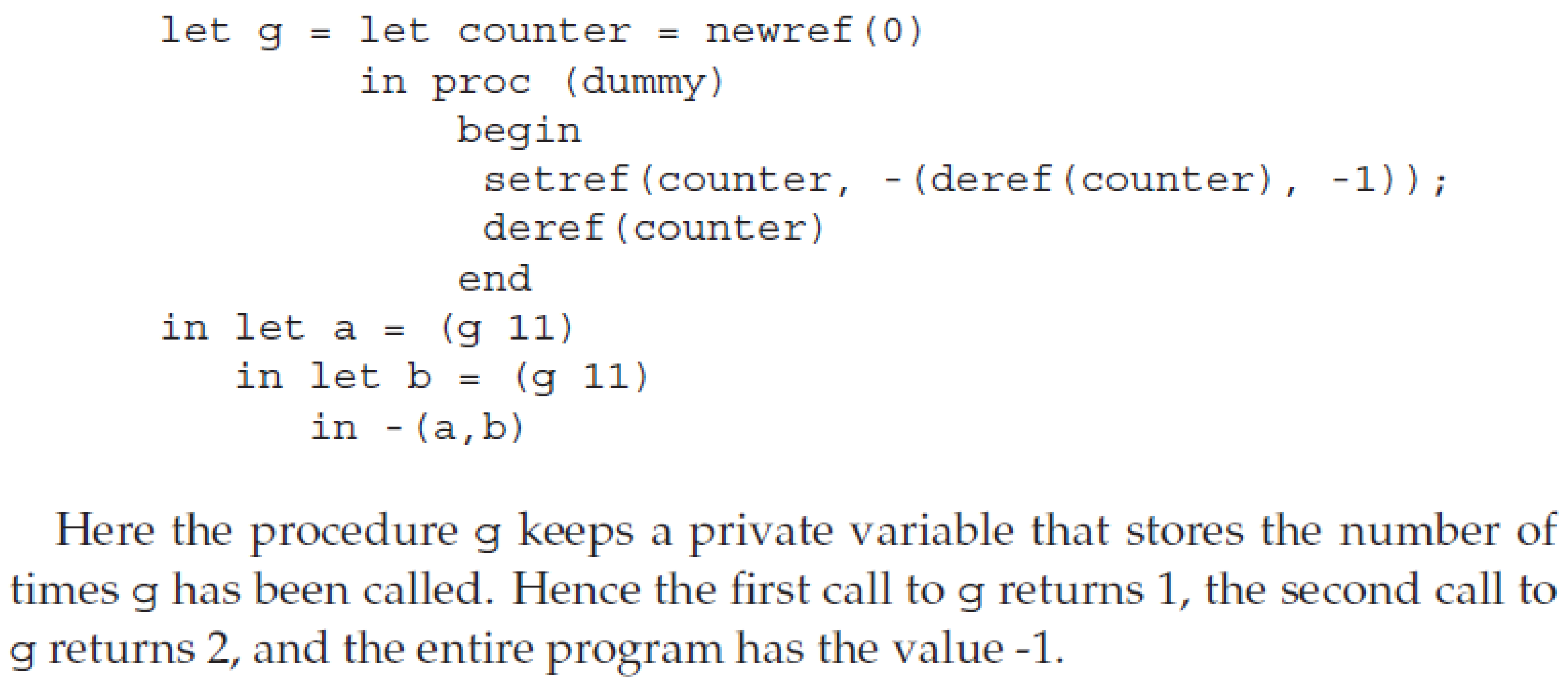
```
let x = newref(0)
in letrec even(dummy)
           = if zero?(deref(x))
             then 1
             else begin
                   setref(x, -(deref(x),1));
                   (odd 888)
                  end
          odd(dummy)
           = if zero?(deref(x))
             then 0
             else begin
                   setref(x, -(deref(x),1));
                   (even 888)
                  end
   in begin setref(x,13); (odd 888) end
```

...

even and odd each take an argument, which they ignore, and return 1 or 0 depending on whether the contents of the location x is even or odd.
They communicate not by passing data explicitly, but by changing the contents of the variable they share.
This program determines whether or not 13 is odd, and therefore returns 1. The procedures even and odd do not refer to their arguments; instead they look at the contents of the location to which x is bound.

This program uses multideclaration letrec (exercise 3.32) and a begin expression (exercise 4.4). A begin expression evaluates its subexpressions in order and returns the value of the last one.

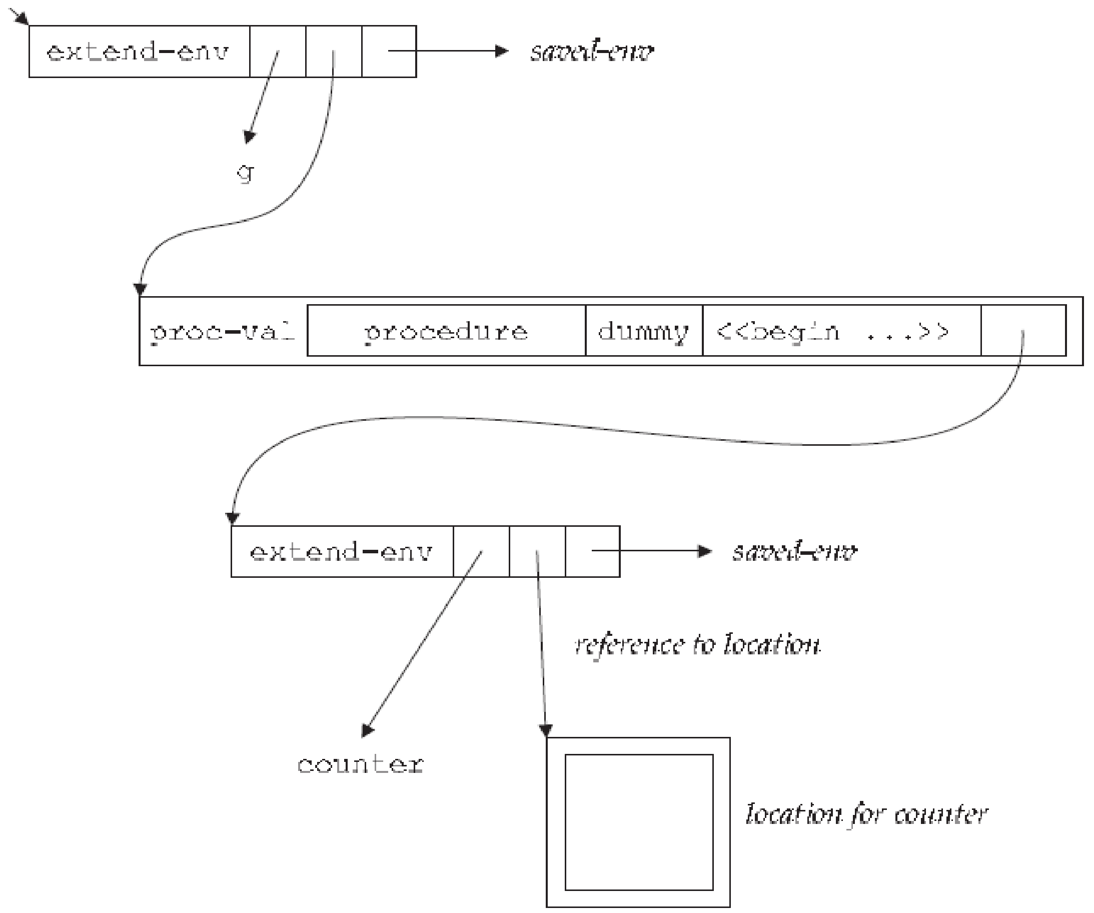
EXPLICIT-REFS: Example2



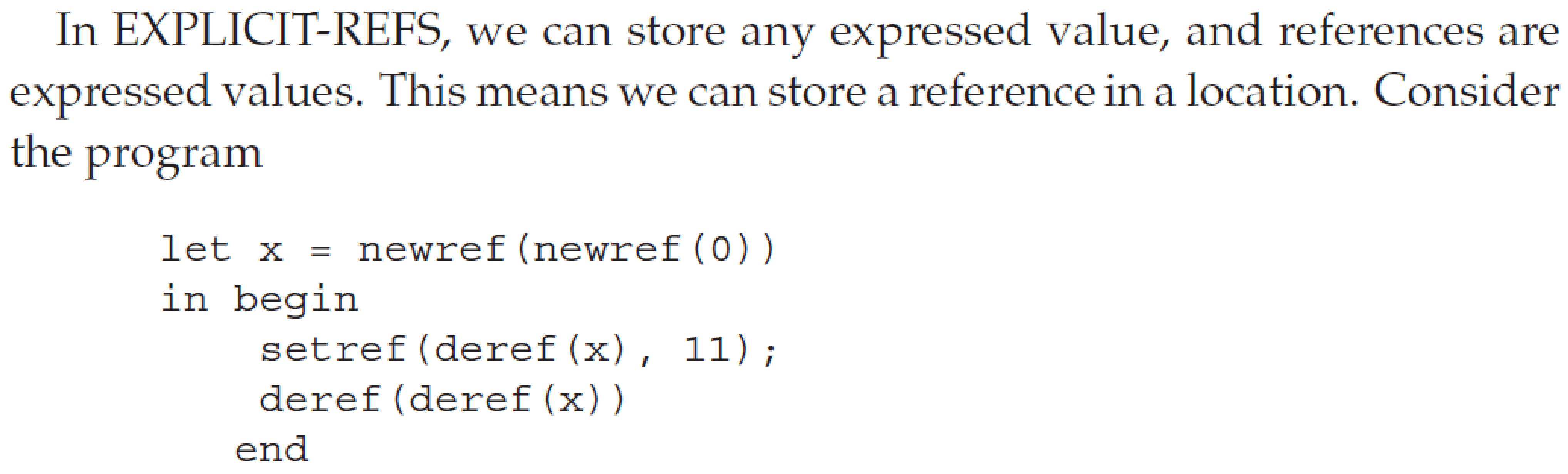
```
let g = let counter = newref(0)
        in proc (dummy)
            begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Here the procedure g keeps a private variable that stores the number of times g has been called. Hence the first call to g returns 1, the second call to g returns 2, and the entire program has the value -1.

Example2: a picture of the environment in which g is bound



EXPLICIT-REFS: Example3



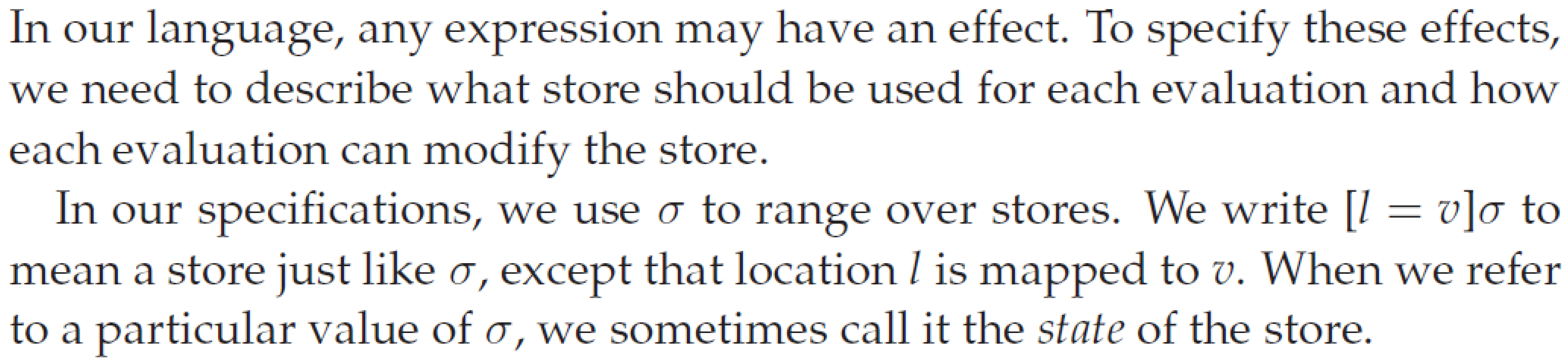
In EXPLICIT-REFS, we can store any expressed value, and references are expressed values. This means we can store a reference in a location. Consider the program

```
let x = newref(newref(0))
in begin
    setref(deref(x), 11);
    deref(deref(x))
   end
```

…

This program allocates a new location containing 0. It then binds `x` to a location containing a reference to the first location. Hence the value of `deref(x)` is a reference to the first location. So when the program evaluates the `setref`, it is the first location that is modified, and the entire program returns 11.

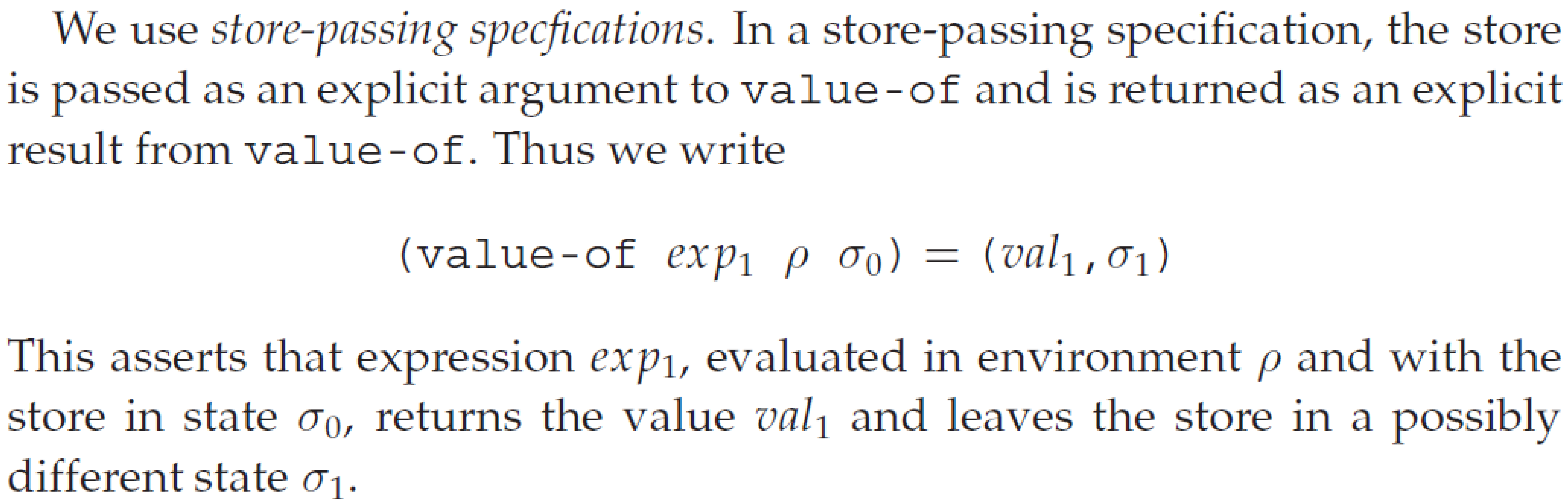
EXPLICIT-REFS: Store-Passing Specifications



In our language, any expression may have an effect. To specify these effects, we need to describe what store should be used for each evaluation and how each evaluation can modify the store.

In our specifications, we use σ to range over stores. We write $[l = v]\sigma$ to mean a store just like σ, except that location l is mapped to v. When we refer to a particular value of σ, we sometimes call it the *state* of the store.

EXPLICIT-REFS: Store-Passing Specifications



We use *store-passing specfications*. In a store-passing specification, the store is passed as an explicit argument to `value-of` and is returned as an explicit result from `value-of`. Thus we write

$$(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)$$

This asserts that expression exp_1, evaluated in environment ρ and with the store in state σ_0, returns the value val_1 and leaves the store in a possibly different state σ_1.

Thus we can specify an effect-free operation like `const-exp` by writing

$$\texttt{(value-of (const-exp } n\texttt{)}\ \rho\ \sigma\texttt{)} = \texttt{(}n\texttt{,}\sigma\texttt{)}$$

showing that the store is unchanged by evaluation of this expression.

The specification for `diff-exp` shows how we specify sequential behavior.

$$\frac{\begin{array}{c}\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\sigma_1\texttt{)} \\ \texttt{(value-of } exp_2\ \rho\ \sigma_1\texttt{)} = \texttt{(}val_2\texttt{,}\sigma_2\texttt{)}\end{array}}{\texttt{(value-of (diff-exp } exp_1\ exp_2\texttt{)}\ \rho\ \sigma_0\texttt{)} = \texttt{(}\lceil\lfloor val_1\rfloor\texttt{-}\lfloor val_2\rfloor\rceil\texttt{,}\sigma_2\texttt{)}}$$

Here we evaluate exp_1 starting with the store in state σ_0. exp_1 returns value val_1, but it might also have some effects that leave the store in state σ_1. We then evaluate exp_2 starting with the store in the state that exp_1 left it, namely σ_1. exp_2 similarly returns a value val_2 and leaves the store in state σ_2. Then the entire expression returns $val_1 - val_2$ without further effect on the store, so it leaves the store in state σ_2.

Let's try a conditional.

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0) \\ \quad = \begin{cases} (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) & \text{if}\ (\texttt{expval->bool}\ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ exp_3\ \rho\ \sigma_1) & \text{if}\ (\texttt{expval->bool}\ val_1) = \texttt{\#f} \end{cases}\end{array}}$$

Starting in state σ_0, an `if-exp` evaluates its test expression exp_1, returning the value val_1 and leaving the store in state σ_1. The result of the entire expression is then either the result of exp_2 or exp_3, each evaluated in the current environment ρ and in the state σ_1 in which exp_1 left the store.

Specifying Operations on Explicit References

In EXPLICIT-REFS, we have three new operations that must be specified: `newref`, `deref`, and `setref`. These are given by the grammar

Expression ::= `newref` (*Expression*)
$\boxed{\texttt{newref-exp (exp1)}}$

Expression ::= `deref` (*Expression*)
$\boxed{\texttt{deref-exp (exp1)}}$

Expression ::= `setref` (*Expression* , *Expression*)
$\boxed{\texttt{setref-exp (exp1 exp2)}}$

…

We can specify the behavior of these operations as follows.

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (val, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{(\texttt{value-of}\ (\texttt{newref-exp}\ exp)\ \rho\ \sigma_0) = ((\texttt{ref-val}\ l), [l{=}val]\sigma_1)}$$

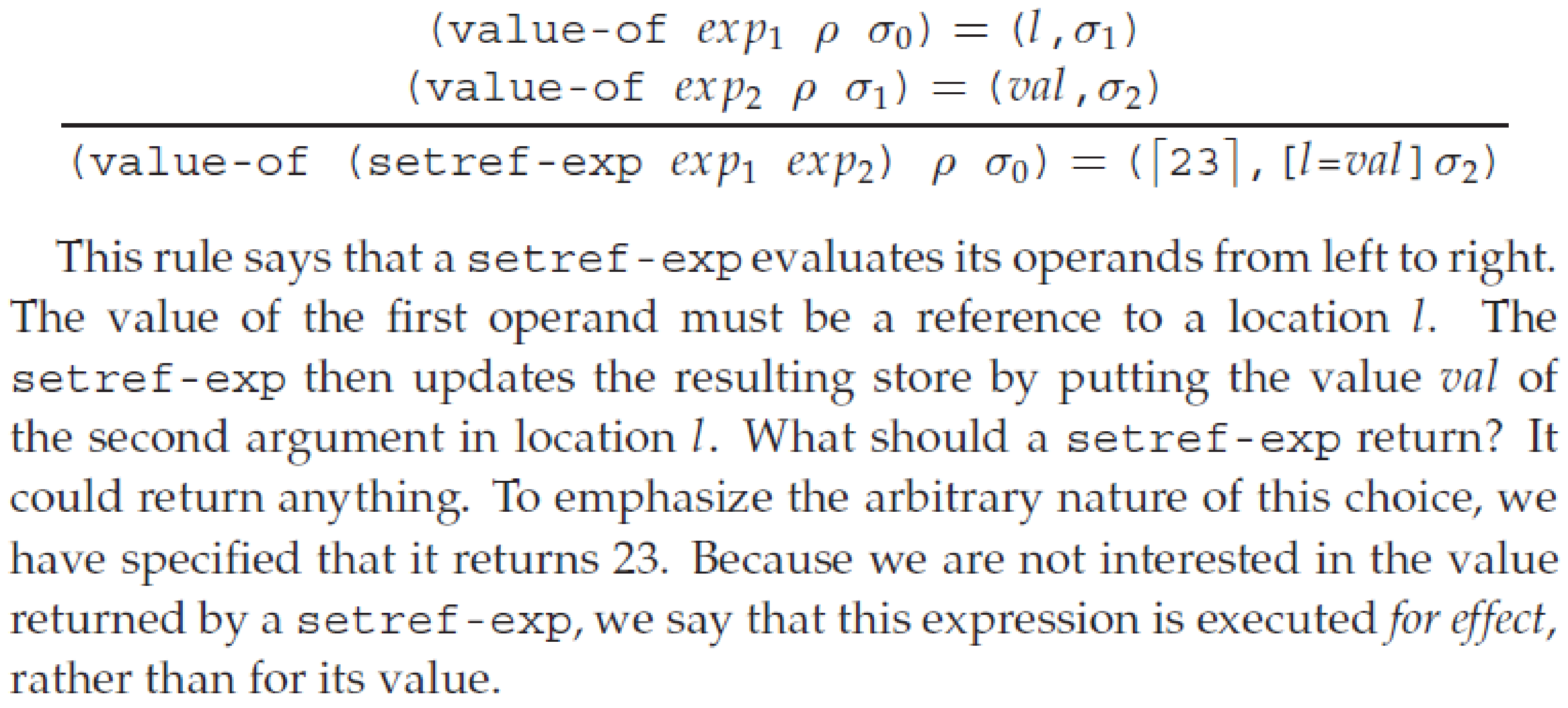
This rule says that `newref-exp` evaluates its operand. It extends the resulting store by allocating a new location l and puts the value val of its argument in that location. Then it returns a reference to a location l that is new. This means that it is not already in the domain of σ_1.

…

$$\frac{(\texttt{value-of}\ exp\ \rho\ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of}\ (\texttt{deref-exp}\ exp)\ \rho\ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

This rule says that a `deref-exp` evaluates its operand, leaving the store in state σ_1. The value of that argument should be a reference to a location l. The `deref-exp` then returns the contents of l in σ_1, without any further change to the store.

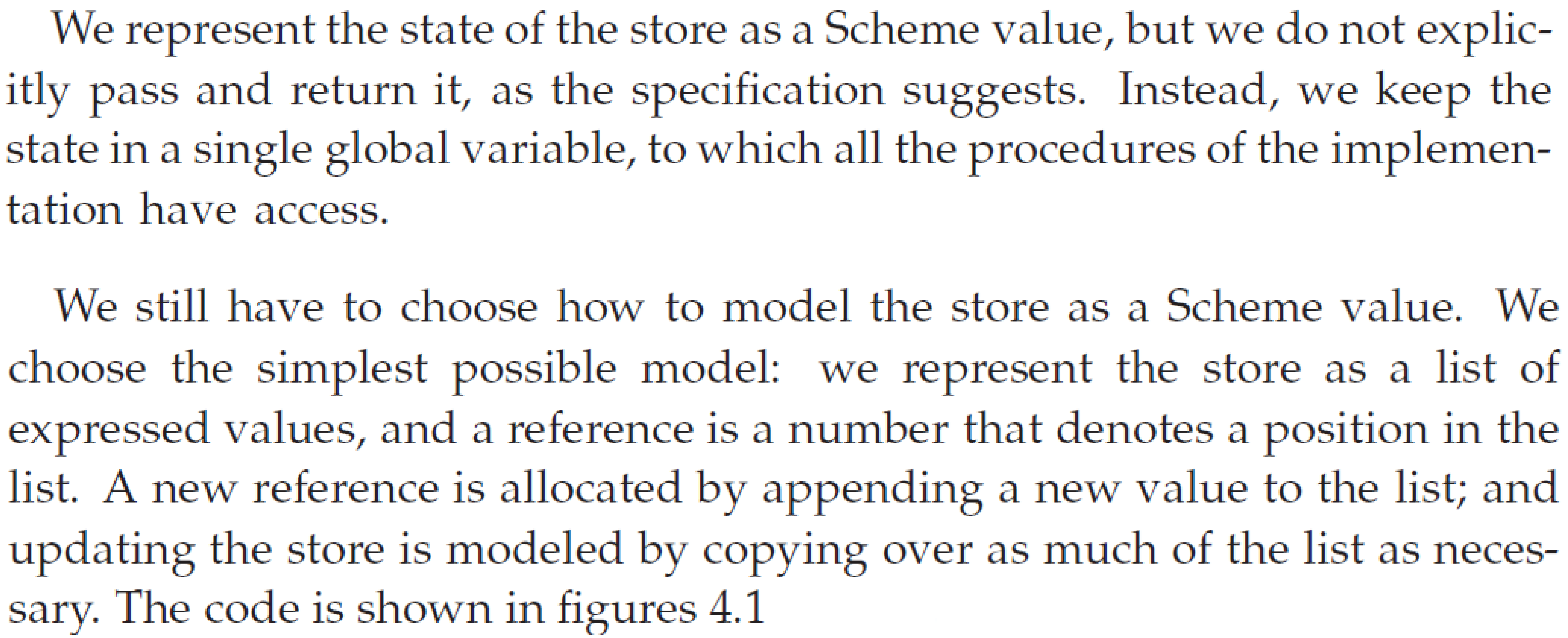
...



$$\frac{\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = (l\texttt{,}\sigma_1\texttt{)} \qquad \texttt{(value-of } exp_2\ \rho\ \sigma_1\texttt{)} = \texttt{(}val\texttt{,}\sigma_2\texttt{)}}{\texttt{(value-of (setref-exp } exp_1\ exp_2\texttt{)}\ \rho\ \sigma_0\texttt{)} = \texttt{(}\lceil 23\rceil\texttt{, [}l\texttt{=}val\texttt{]}\sigma_2\texttt{)}}$$

This rule says that a `setref-exp` evaluates its operands from left to right. The value of the first operand must be a reference to a location l. The `setref-exp` then updates the resulting store by putting the value *val* of the second argument in location l. What should a `setref-exp` return? It could return anything. To emphasize the arbitrary nature of this choice, we have specified that it returns 23. Because we are not interested in the value returned by a `setref-exp`, we say that this expression is executed *for effect*, rather than for its value.

EXPLICIT-REFS: Implementation



We represent the state of the store as a Scheme value, but we do not explicitly pass and return it, as the specification suggests. Instead, we keep the state in a single global variable, to which all the procedures of the implementation have access.

We still have to choose how to model the store as a Scheme value. We choose the simplest possible model: we represent the store as a list of expressed values, and a reference is a number that denotes a position in the list. A new reference is allocated by appending a new value to the list; and updating the store is modeled by copying over as much of the list as necessary. The code is shown in figures 4.1

empty-store : () $\rightarrow$ *Sto*
```
(define empty-store
  (lambda () '()))
```

usage: A Scheme variable containing the current state of the store. Initially set to a dummy value.
```
(define the-store 'uninitialized)
```

get-store : () $\rightarrow$ *Sto*
```
(define get-store
  (lambda () the-store))
```

initialize-store! : () $\rightarrow$ *Unspecified*
usage: (initialize-store!) sets the-store to the empty store
```
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))
```

Figure 4.1 A naive model of the store

reference? : *SchemeVal* → *Bool*
```
(define reference?
  (lambda (v)
    (integer? v)))
```

newref : *ExpVal* → *Ref*
```
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref)))
```

deref : *Ref* → *ExpVal*
```
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

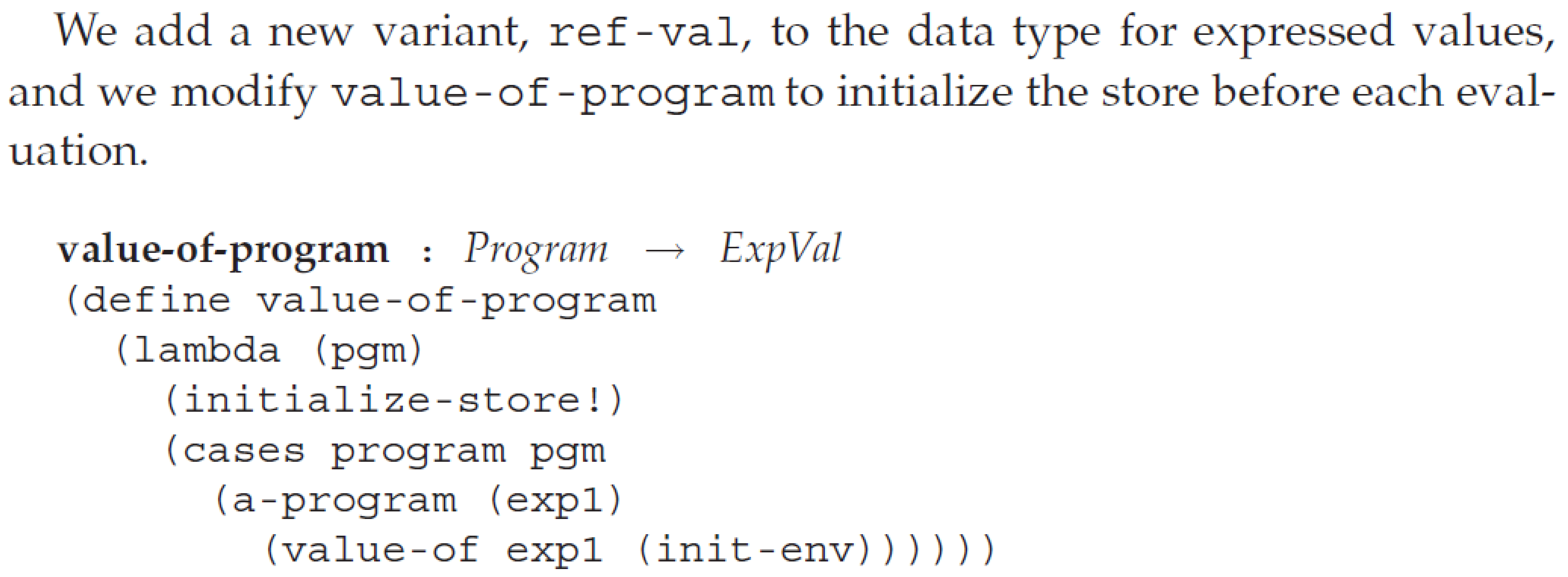
Figure 4.1 A naive model of the store

setref! : *Ref* × *ExpVal* → *Unspecified*
usage: sets the-store to a state like the original, but with position ref containing val.

```
(define setref!
  (lambda (ref val)
    (set! the-store
      (letrec
        ((setref-inner
           usage: returns a list like store1, except that
           position ref1 contains val.
           (lambda (store1 ref1)
             (cond
               ((null? store1)
                (report-invalid-reference ref the-store))
               ((zero? ref1)
                (cons val (cdr store1)))
               (else
                 (cons
                   (car store1)
                   (setref-inner
                     (cdr store1) (- ref1 1))))))))
        (setref-inner the-store ref)))))
```

Figure 4.1 A naive model of the store

EXPLICIT-REFS: Implementation

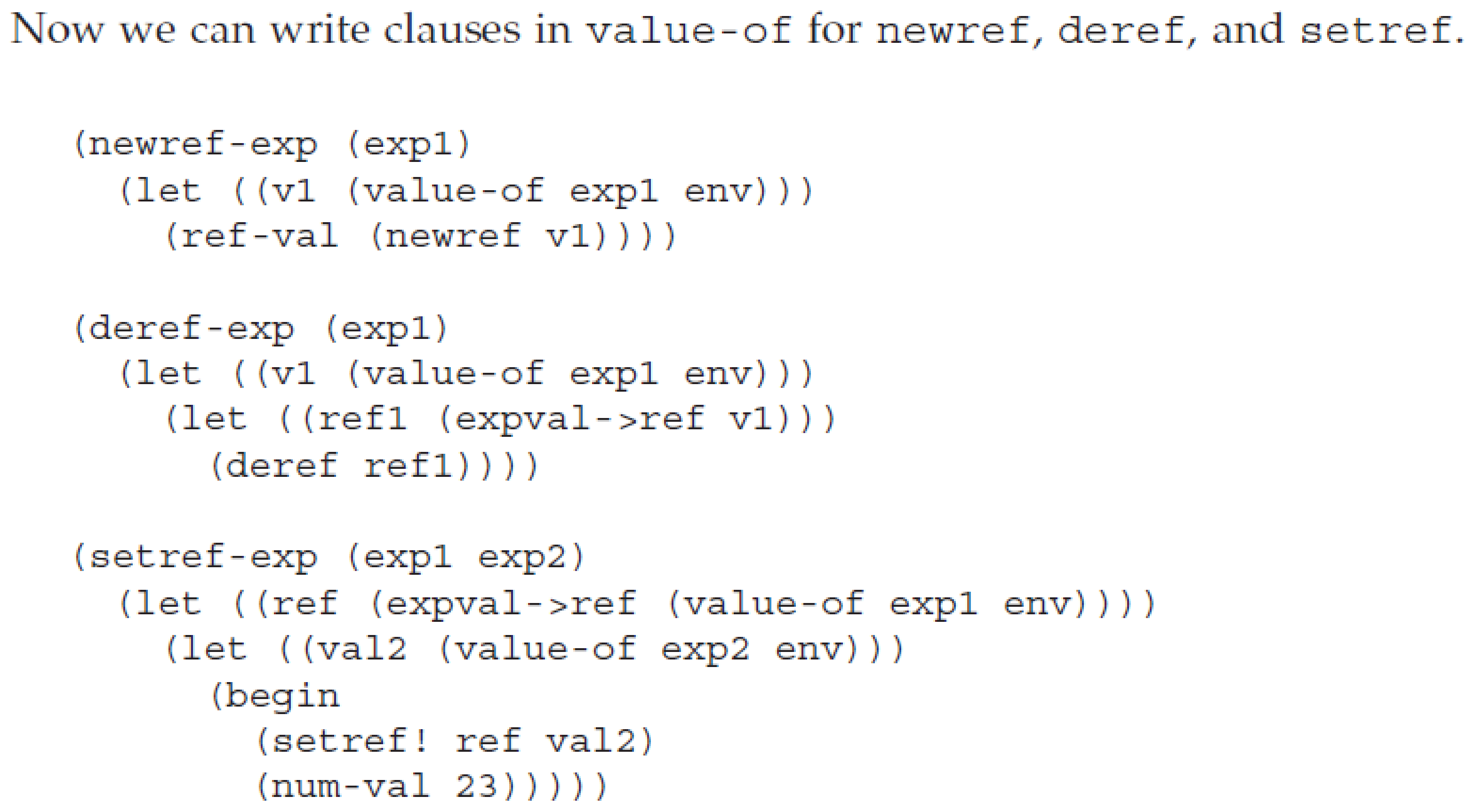


We add a new variant, `ref-val`, to the data type for expressed values, and we modify `value-of-program` to initialize the store before each evaluation.

value-of-program : *Program* $\rightarrow$ *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

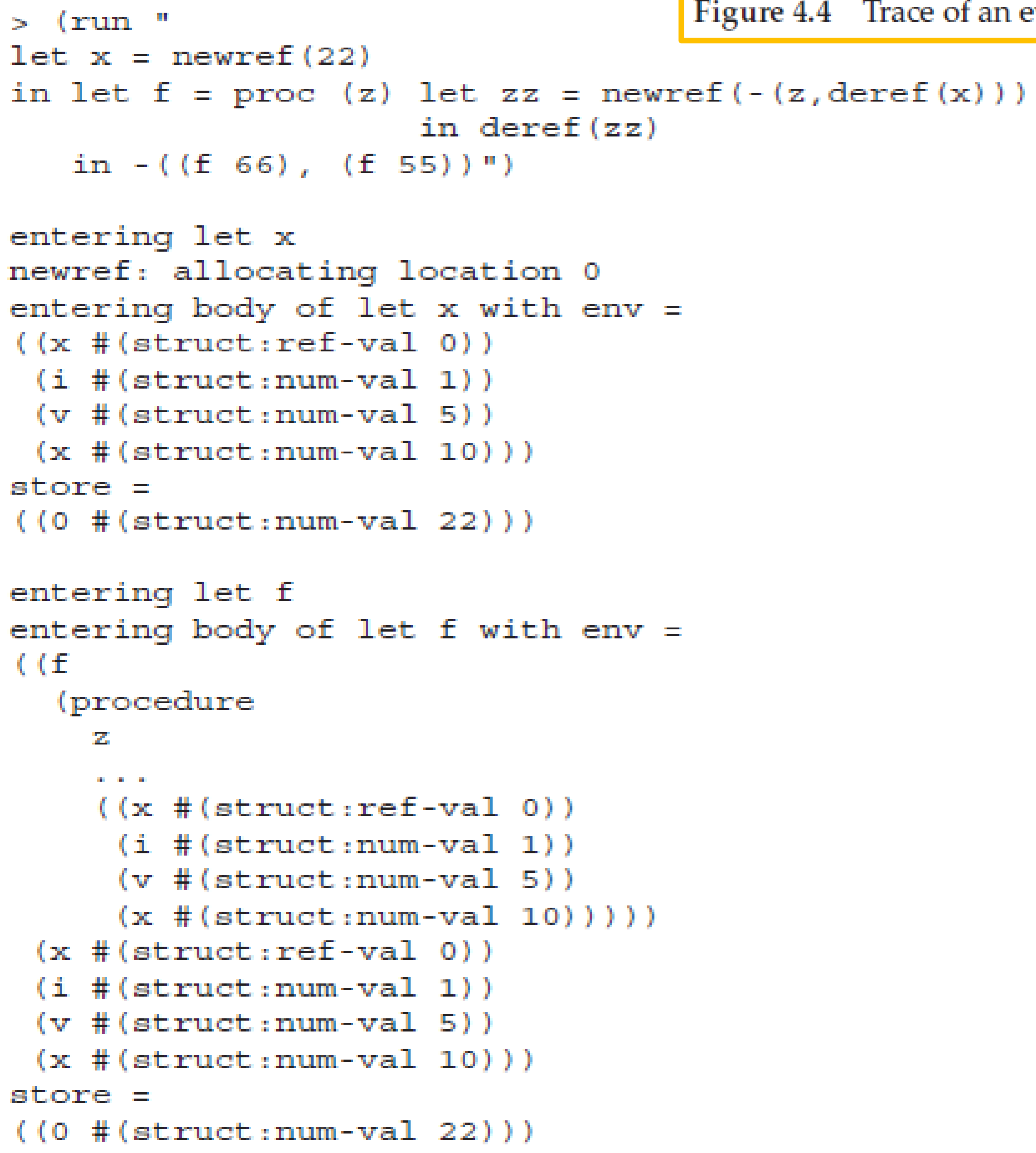
EXPLICIT-REFS: Implementation



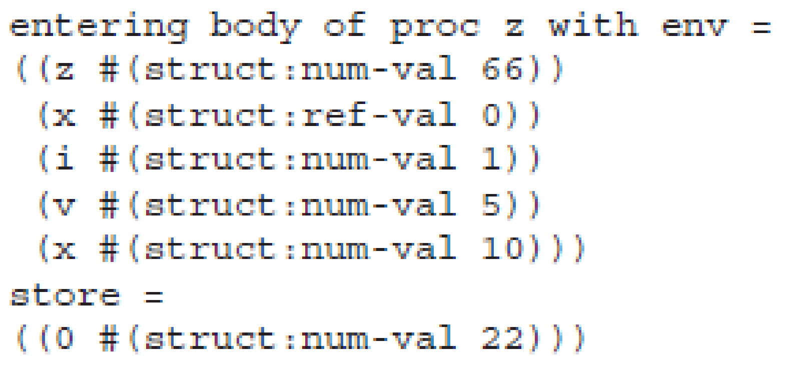
Now we can write clauses in `value-of` for `newref`, `deref`, and `setref`.

```
(newref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (ref-val (newref v1))))

(deref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (let ((ref1 (expval->ref v1)))
      (deref ref1))))

(setref-exp (exp1 exp2)
  (let ((ref (expval->ref (value-of exp1 env))))
    (let ((val2 (value-of exp2 env)))
      (begin
        (setref! ref val2)
        (num-val 23)))))
```

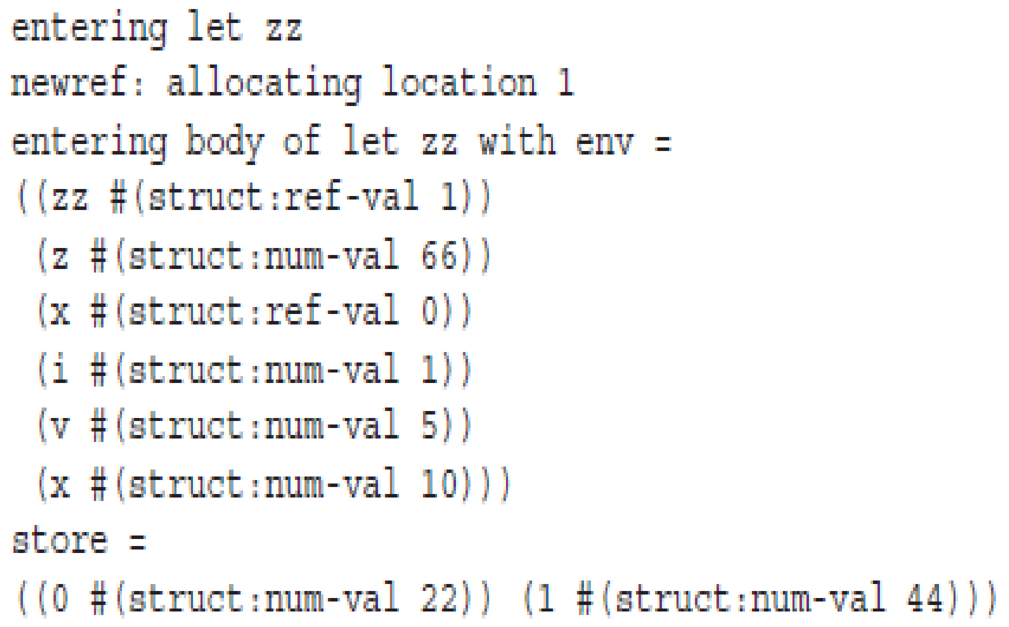
Figure 4.4 Trace of an evaluation in EXPLICIT-REFS.



```
> (run "
let x = newref(22)
in let f = proc (z) let zz = newref(-(z,deref(x)))
                    in deref(zz)
   in -((f 66), (f 55))")

entering let x
newref: allocating location 0
entering body of let x with env =
((x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let f
entering body of let f with env =
((f
  (procedure
    z
    ...
    ((x #(struct:ref-val 0))
     (i #(struct:num-val 1))
     (v #(struct:num-val 5))
     (x #(struct:num-val 10)))))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))
```



```
entering body of proc z with env =
((z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))
```

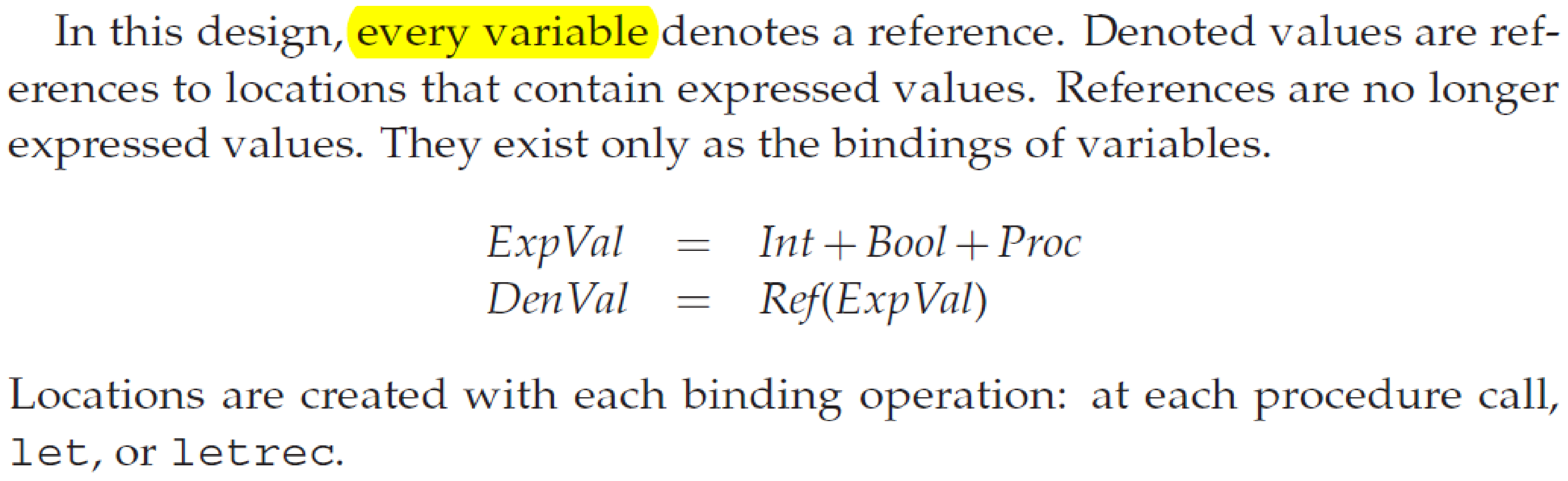


```
entering let zz
newref: allocating location 1
entering body of let zz with env =
((zz #(struct:ref-val 1))
 (z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))
```

Figure 4.4 Trace of an evaluation in EXPLICIT-REFS.

```
entering body of proc z with env =
((z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

entering let zz
newref: allocating location 2
entering body of let zz with env =
((zz #(struct:ref-val 2))
 (z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22))
 (1 #(struct:num-val 44))
 (2 #(struct:num-val 33)))

#(struct:num-val 11)
>
```

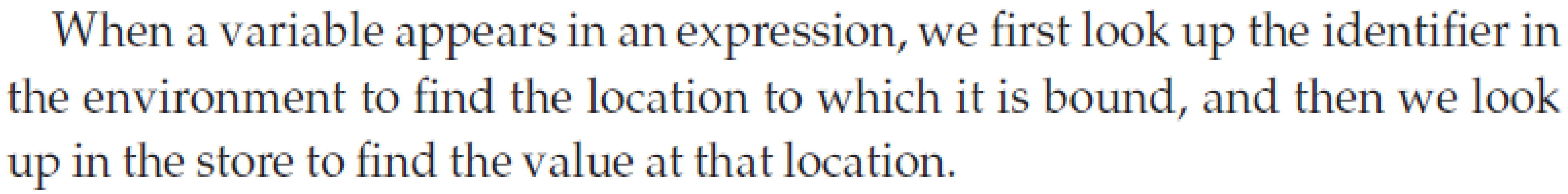
IMPLICIT-REFS: A Language with Implicit References



In this design, every variable denotes a reference. Denoted values are references to locations that contain expressed values. References are no longer expressed values. They exist only as the bindings of variables.

$$
\begin{aligned}
\mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} \\
\mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal})
\end{aligned}
$$

Locations are created with each binding operation: at each procedure call, `let`, or `letrec`.



When a variable appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location.

IMPLICIT-REFS: A Language with Implicit References

When a variable appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location. Hence we have a “two-level” system for `var-exp`.

The contents of a location can be changed by a `set` expression. We use the syntax

$$\textit{Expression} ::= \texttt{set}\ \ \textit{Identifier}\ \ \texttt{=}\ \ \textit{Expression}$$
$$\boxed{\texttt{assign-exp (var exp1)}}$$

Here the *Identifier* is not part of an expression, so it does not get dereferenced. In this design, we say that variables are *mutable*, meaning changeable.

```
let x = 0
in letrec even(dummy)
           = if zero?(x)
             then 1
             else begin
                   set x = -(x,1);
                   (odd 888)
                  end
          odd(dummy)
           = if zero?(x)
             then 0
             else begin
                   set x = -(x,1);
                   (even 888)
                  end
   in begin set x = 13; (odd -888) end

let g = let count = 0
        in proc (dummy)
            begin
             set count = -(count,-1);
             count
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Figure 4.7 odd and even in IMPLICIT-REFS

IMPLICIT-REFS

This design is called *call-by-value*, or *implicit references*.

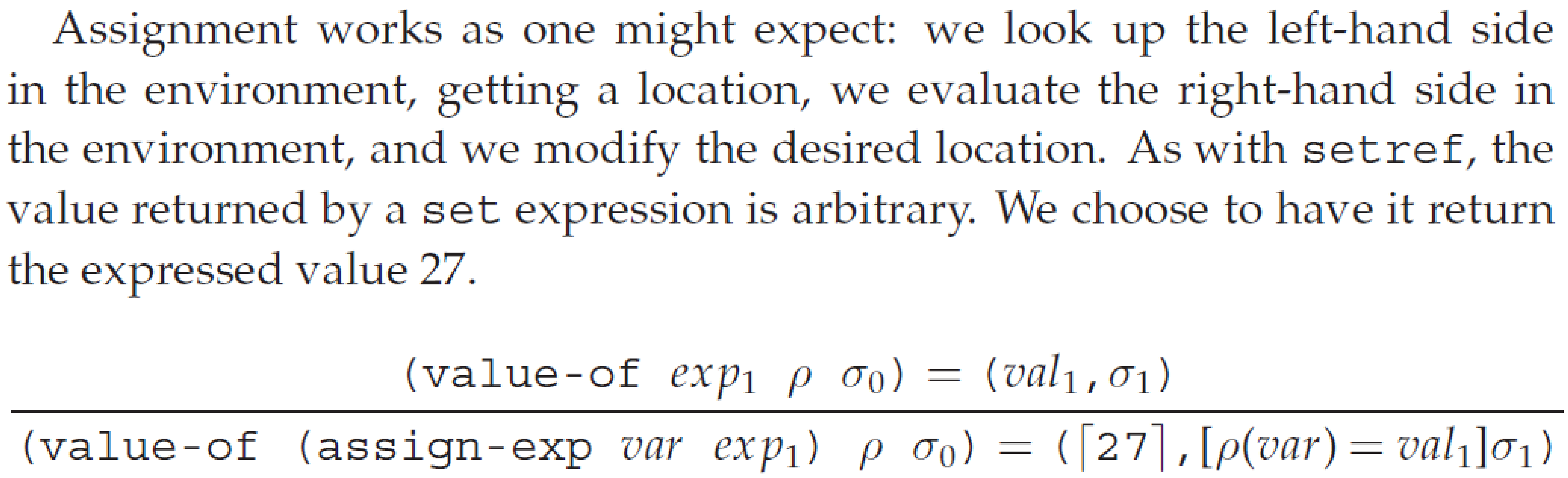
Because references are no longer expressed values, we can't
make chains of references, as we did in the last example in section 4.2.

IMPLICIT-REFS: Specification

We can write the rules for dereference and `set` easily. The environment now always binds variables to locations, so when a variable appears as an expression, we need to dereference it:

$$\texttt{(value-of (var-exp } \mathit{var}\texttt{)}\ \rho\ \sigma\texttt{)} = (\sigma(\rho(\mathit{var})), \sigma)$$

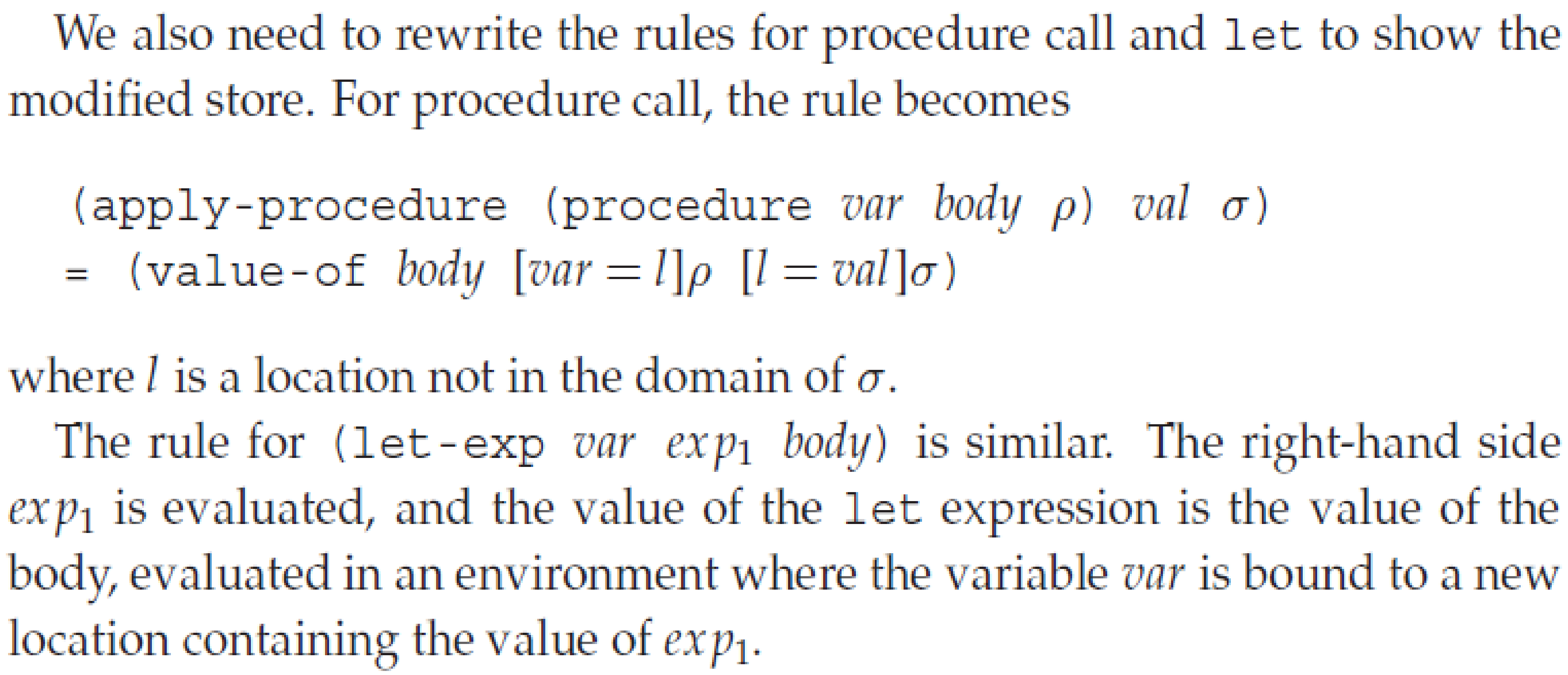
IMPLICIT-REFS: Specification



Assignment works as one might expect: we look up the left-hand side in the environment, getting a location, we evaluate the right-hand side in the environment, and we modify the desired location. As with `setref`, the value returned by a `set` expression is arbitrary. We choose to have it return the expressed value 27.

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ (\texttt{assign-exp}\ var\ exp_1)\ \rho\ \sigma_0) = (\lceil 27 \rceil, [\rho(var) = val_1]\sigma_1)}$$

IMPLICIT-REFS: Specification



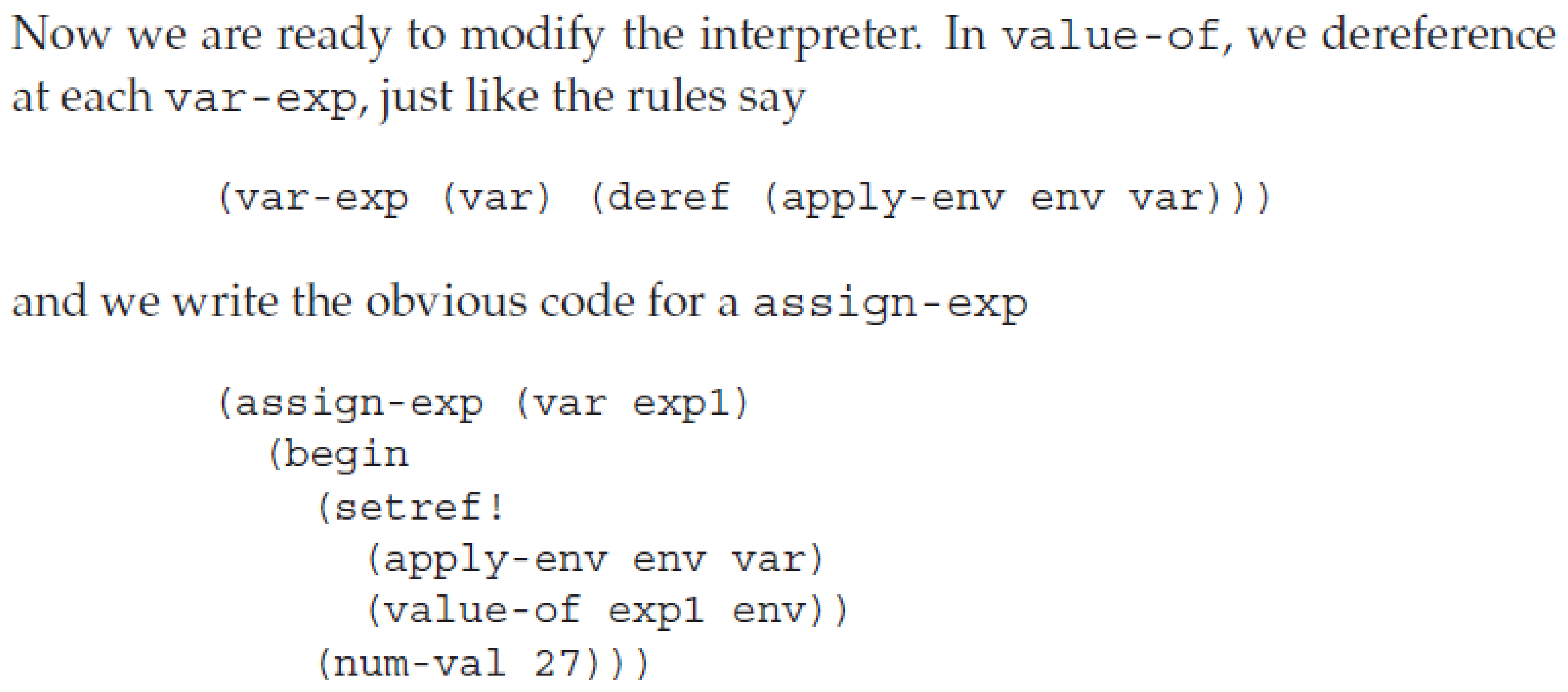
We also need to rewrite the rules for procedure call and `let` to show the modified store. For procedure call, the rule becomes

$$
\begin{array}{l}
\texttt{(apply-procedure (procedure } \mathit{var}\ \mathit{body}\ \rho\texttt{)}\ \mathit{val}\ \sigma\texttt{)} \\
\texttt{= (value-of } \mathit{body}\ [\mathit{var}=l]\rho\ [l=\mathit{val}]\sigma\texttt{)}
\end{array}
$$

where l is a location not in the domain of σ.

The rule for `(let-exp` var exp_1 body`)` is similar. The right-hand side exp_1 is evaluated, and the value of the `let` expression is the value of the body, evaluated in an environment where the variable var is bound to a new location containing the value of exp_1.

IMPLICIT-REFS: The Implementation



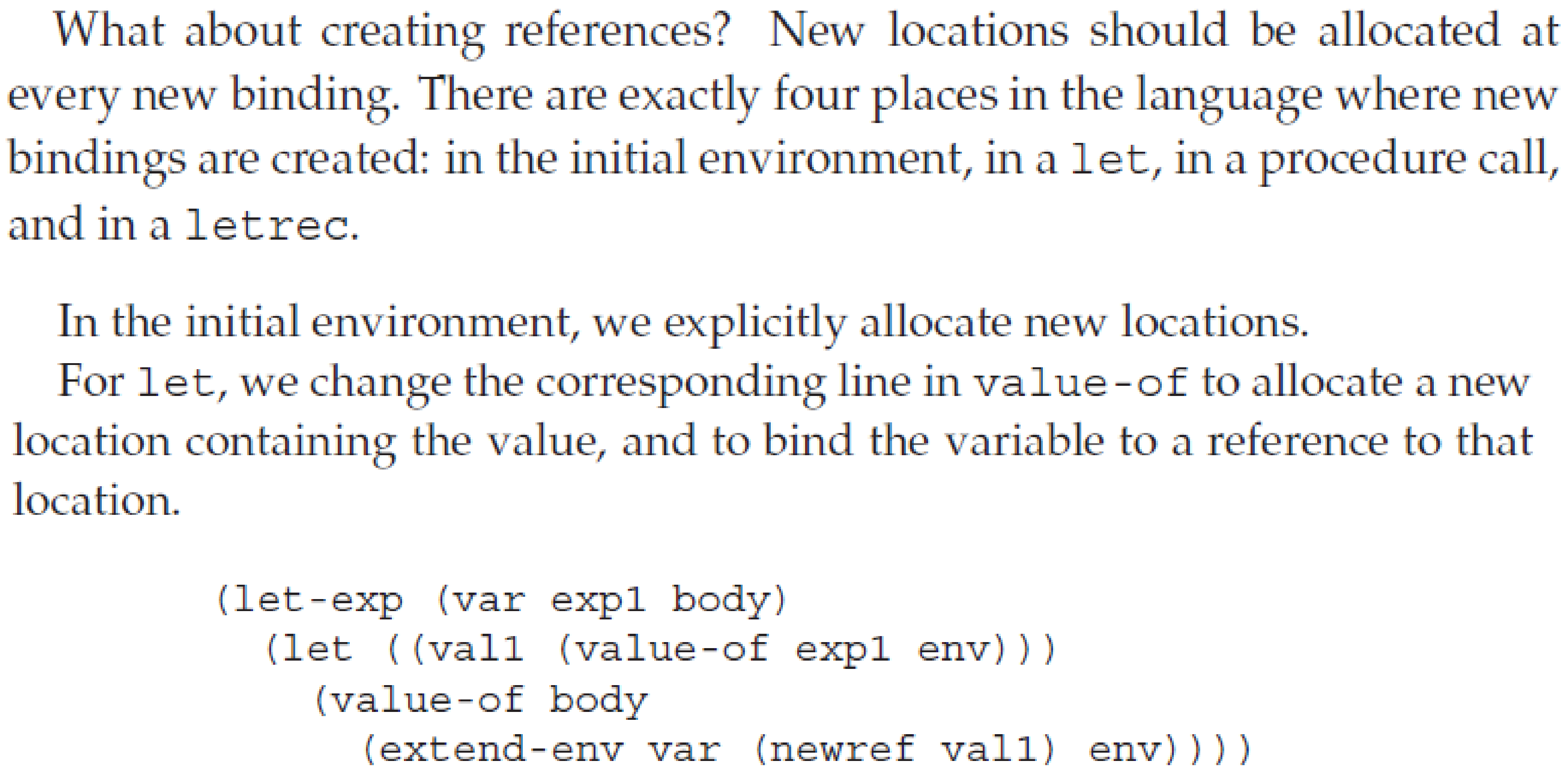
Now we are ready to modify the interpreter. In `value-of`, we dereference at each `var-exp`, just like the rules say

```
(var-exp (var) (deref (apply-env env var)))
```

and we write the obvious code for a `assign-exp`

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

IMPLICIT-REFS: The Implementation



What about creating references? New locations should be allocated at every new binding. There are exactly four places in the language where new bindings are created: in the initial environment, in a `let`, in a procedure call, and in a `letrec`.

In the initial environment, we explicitly allocate new locations.

For `let`, we change the corresponding line in `value-of` to allocate a new location containing the value, and to bind the variable to a reference to that location.

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

IMPLICIT-REFS: The Implementation

For a procedure call, we similarly change `apply-procedure` to call `newref`.

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

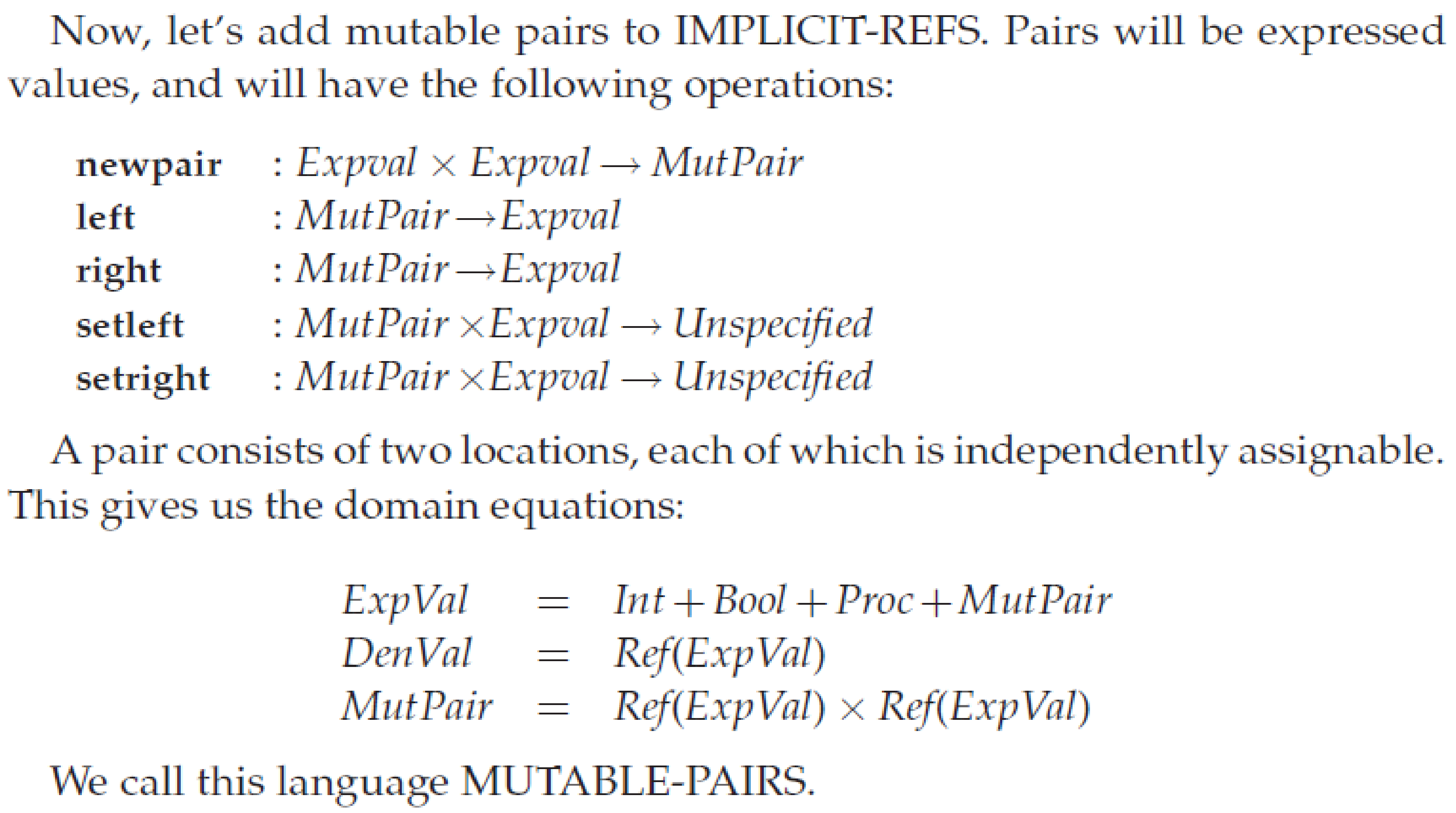
```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

Figure 4.8 Sample evaluation in IMPLICIT-REFS

```
> (run  "
let f = proc (x) proc (y)
         begin
          set x = -(x,-1);
          -(x,y)
         end
in ((f 44) 33)")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))
```

```
newref: allocating location 4
entering body of proc x with env =
((x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44)))

newref: allocating location 5
entering body of proc y with env =
((y 5) (x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44))
 (5 #(struct:num-val 33)))

#(struct:num-val 12)
```

MUTABLE-PAIRS: A Language with Mutable Pairs



Now, let's add mutable pairs to IMPLICIT-REFS. Pairs will be expressed values, and will have the following operations:

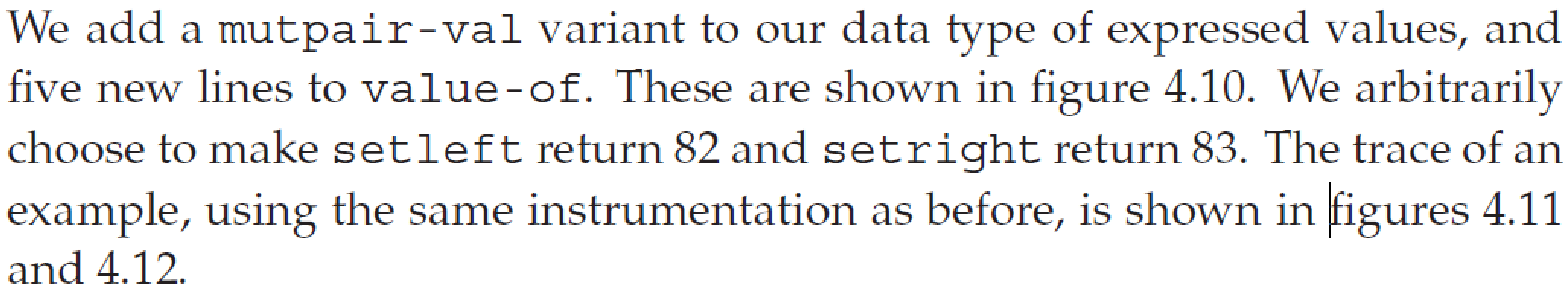
newpair	: $Expval \times Expval \rightarrow MutPair$
left	: $MutPair \rightarrow Expval$
right	: $MutPair \rightarrow Expval$
setleft	: $MutPair \times Expval \rightarrow Unspecified$
setright	: $MutPair \times Expval \rightarrow Unspecified$

A pair consists of two locations, each of which is independently assignable. This gives us the domain equations:

$$\begin{aligned} ExpVal &= Int + Bool + Proc + MutPair \\ DenVal &= Ref(ExpVal) \\ MutPair &= Ref(ExpVal) \times Ref(ExpVal) \end{aligned}$$

We call this language MUTABLE-PAIRS.

MUTABLE-PAIRS: Implementation



We add a `mutpair-val` variant to our data type of expressed values, and five new lines to `value-of`. These are shown in figure 4.10. We arbitrarily choose to make `setleft` return 82 and `setright` return 83. The trace of an example, using the same instrumentation as before, is shown in figures 4.11 and 4.12.

Figure 4.9 Naive implementation of mutable pairs

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))
```

make-pair : $ExpVal \times ExpVal \rightarrow MutPair$

```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : $MutPair \rightarrow ExpVal$

```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

right : $MutPair \rightarrow ExpVal$

```
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))
```

setleft : $MutPair \times ExpVal \rightarrow Unspecified$

```
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))
```

setright : $MutPair \times ExpVal \rightarrow Unspecified$

```
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

Figure 4.10 Integrating mutable pairs into the interpreter

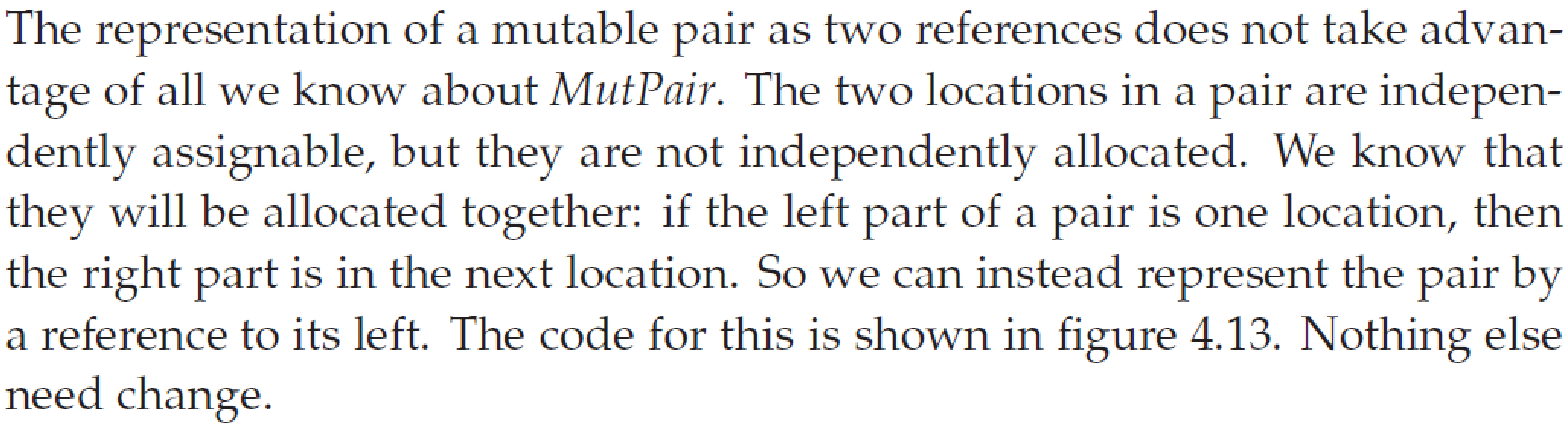
```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))

(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))
```

```
(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

Figure 4.11 Trace of evaluation in MUTABLE-PAIRS

```
> (run "let glo = pair(11,22)
in let f = proc (loc)
            let d1 = setright(loc, left(loc))
            in let d2 = setleft(glo, 99)
            in -(left(loc),right(loc))
in (f glo)")
;; allocating cells for init-env
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let glo
;; allocating cells for the pair
newref: allocating location 3
newref: allocating location 4
;; allocating cell for glo
newref: allocating location 5
entering body of let glo with env =
((glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4))))
entering let f
;; allocating cell for f
newref: allocating location 6
entering body of let f with env =
((f 6) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2)))))
;; allocating cell for loc
newref: allocating location 7
entering body of proc loc with env =
((loc 7) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2))))
 (7 #(struct:mutpair-val #(struct:a-pair 3 4))))

#(struct:num-val 88)
>
```

Another Representation of Mutable Pairs



The representation of a mutable pair as two references does not take advantage of all we know about *MutPair*. The two locations in a pair are independently assignable, but they are not independently allocated. We know that they will be allocated together: if the left part of a pair is one location, then the right part is in the next location. So we can instead represent the pair by a reference to its left. The code for this is shown in figure 4.13. Nothing else need change.

Figure 4.13 Alternate representation of mutable pairs

mutpair? : $SchemeVal \rightarrow Bool$

```
(define mutpair?
  (lambda (v)
    (reference? v)))
```

make-pair : $ExpVal \times ExpVal \rightarrow MutPair$

```
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))
```

left : $MutPair \rightarrow ExpVal$

```
(define left
  (lambda (p)
    (deref p)))
```

right : $MutPair \rightarrow ExpVal$

```
(define right
  (lambda (p)
    (deref (+ 1 p))))
```

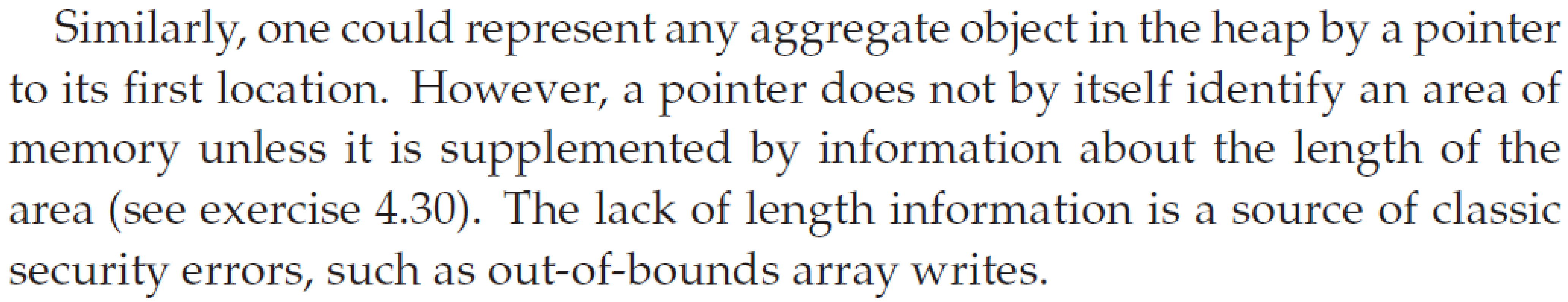
setleft : $MutPair \times ExpVal \rightarrow Unspecified$

```
(define setleft
  (lambda (p val)
    (setref! p val)))
```

setright : $MutPair \times ExpVal \rightarrow Unspecified$

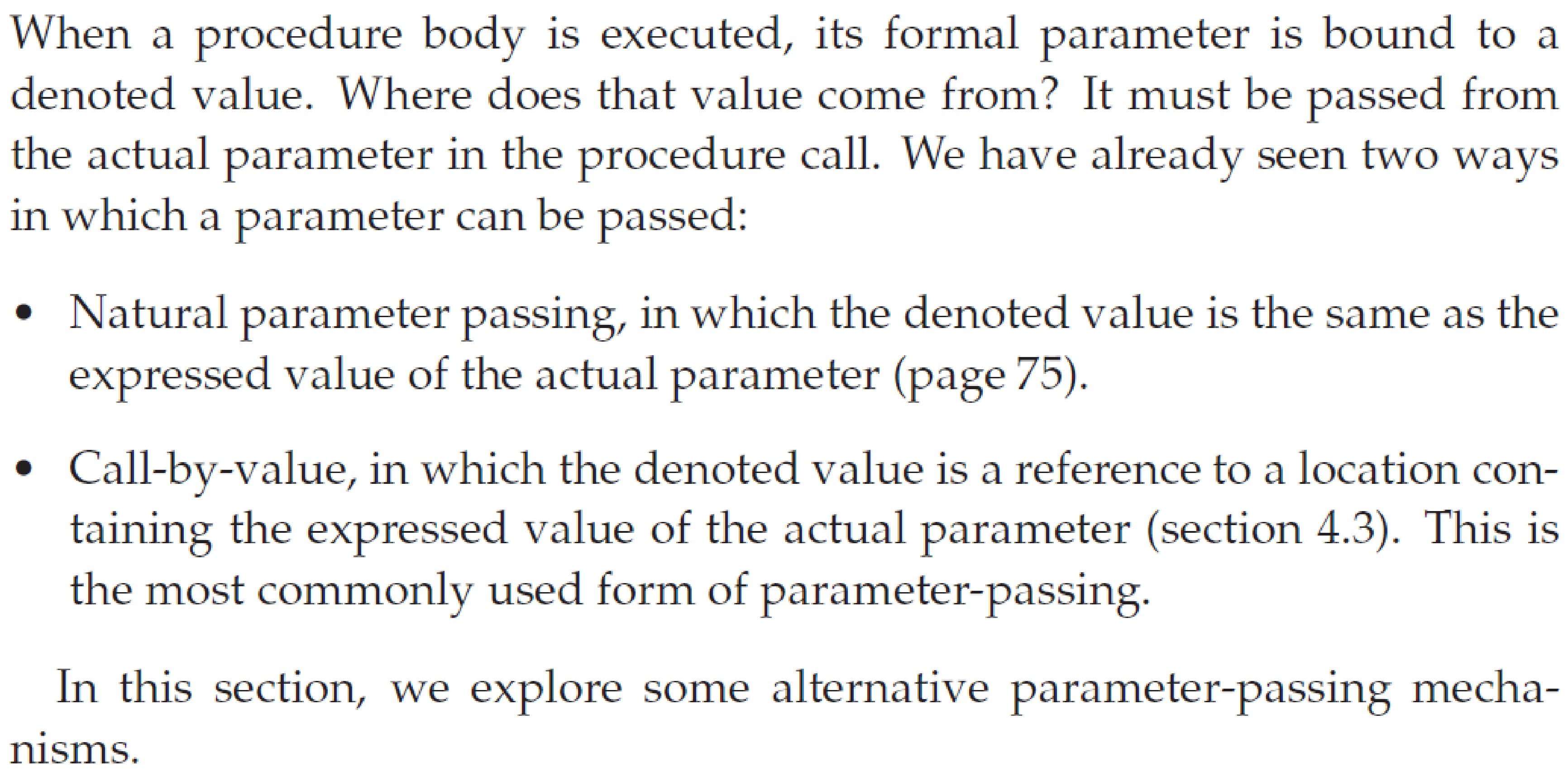
```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

…



Similarly, one could represent any aggregate object in the heap by a pointer to its first location. However, a pointer does not by itself identify an area of memory unless it is supplemented by information about the length of the area (see exercise 4.30). The lack of length information is a source of classic security errors, such as out-of-bounds array writes.

Parameter-Passing Variations

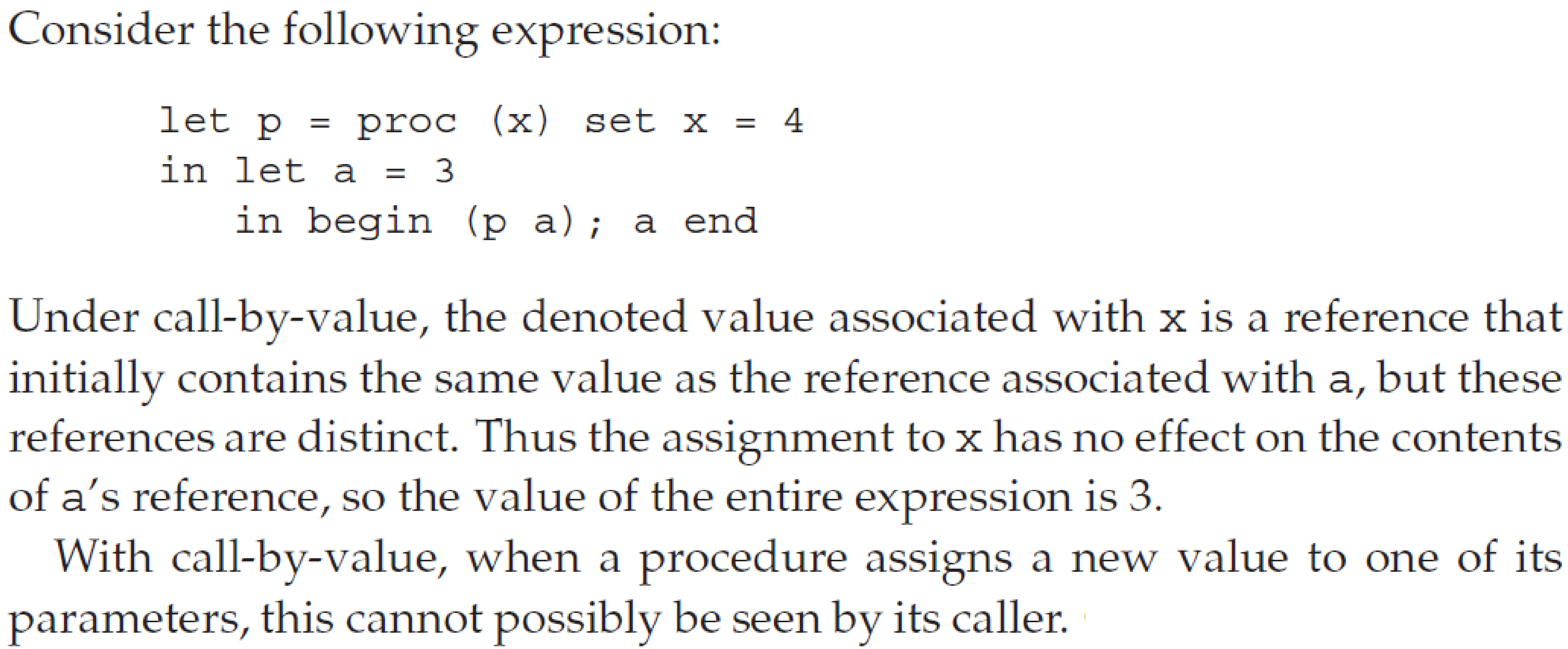


When a procedure body is executed, its formal parameter is bound to a denoted value. Where does that value come from? It must be passed from the actual parameter in the procedure call. We have already seen two ways in which a parameter can be passed:

- Natural parameter passing, in which the denoted value is the same as the expressed value of the actual parameter (page 75).
- Call-by-value, in which the denoted value is a reference to a location containing the expressed value of the actual parameter (section 4.3). This is the most commonly used form of parameter-passing.

In this section, we explore some alternative parameter-passing mechanisms.

Parameter-Passing Variations: CALL-BY-REFERENCE



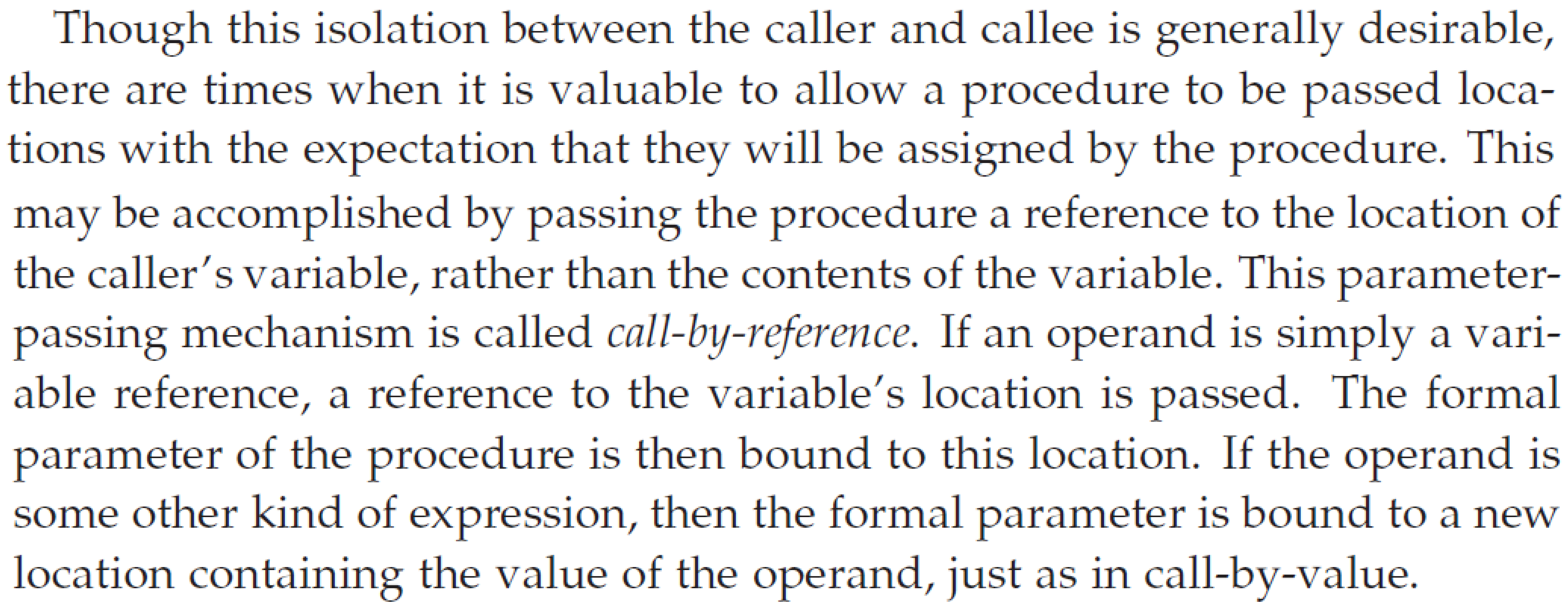
Consider the following expression:

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Under call-by-value, the denoted value associated with x is a reference that initially contains the same value as the reference associated with a, but these references are distinct. Thus the assignment to x has no effect on the contents of a's reference, so the value of the entire expression is 3.

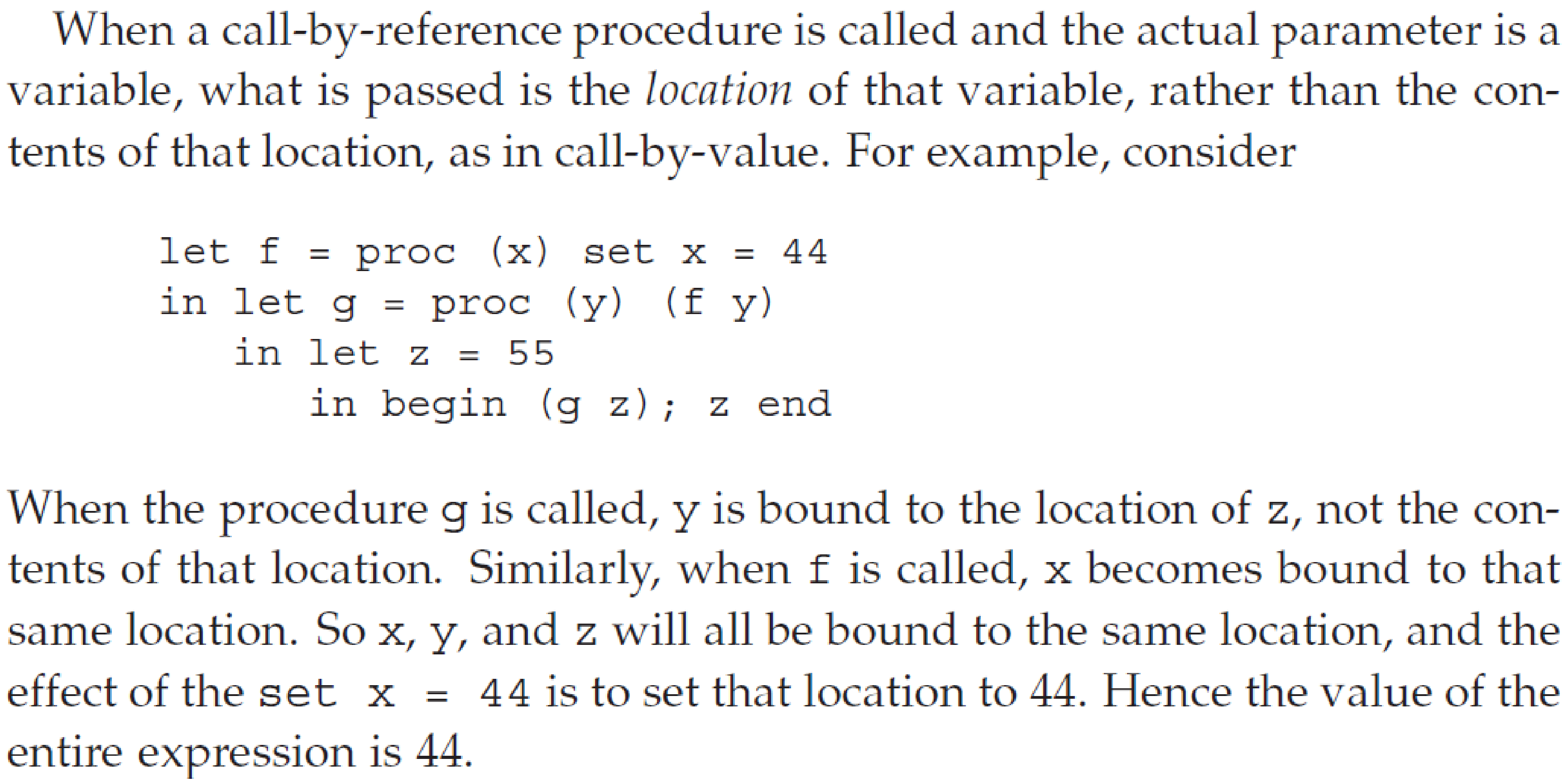
With call-by-value, when a procedure assigns a new value to one of its parameters, this cannot possibly be seen by its caller.

Parameter-Passing Variations: CALL-BY-REFERENCE



Though this isolation between the caller and callee is generally desirable, there are times when it is valuable to allow a procedure to be passed locations with the expectation that they will be assigned by the procedure. This may be accomplished by passing the procedure a reference to the location of the caller's variable, rather than the contents of the variable. This parameter-passing mechanism is called *call-by-reference*. If an operand is simply a variable reference, a reference to the variable's location is passed. The formal parameter of the procedure is then bound to this location. If the operand is some other kind of expression, then the formal parameter is bound to a new location containing the value of the operand, just as in call-by-value.

Parameter-Passing Variations: CALL-BY-REFERENCE

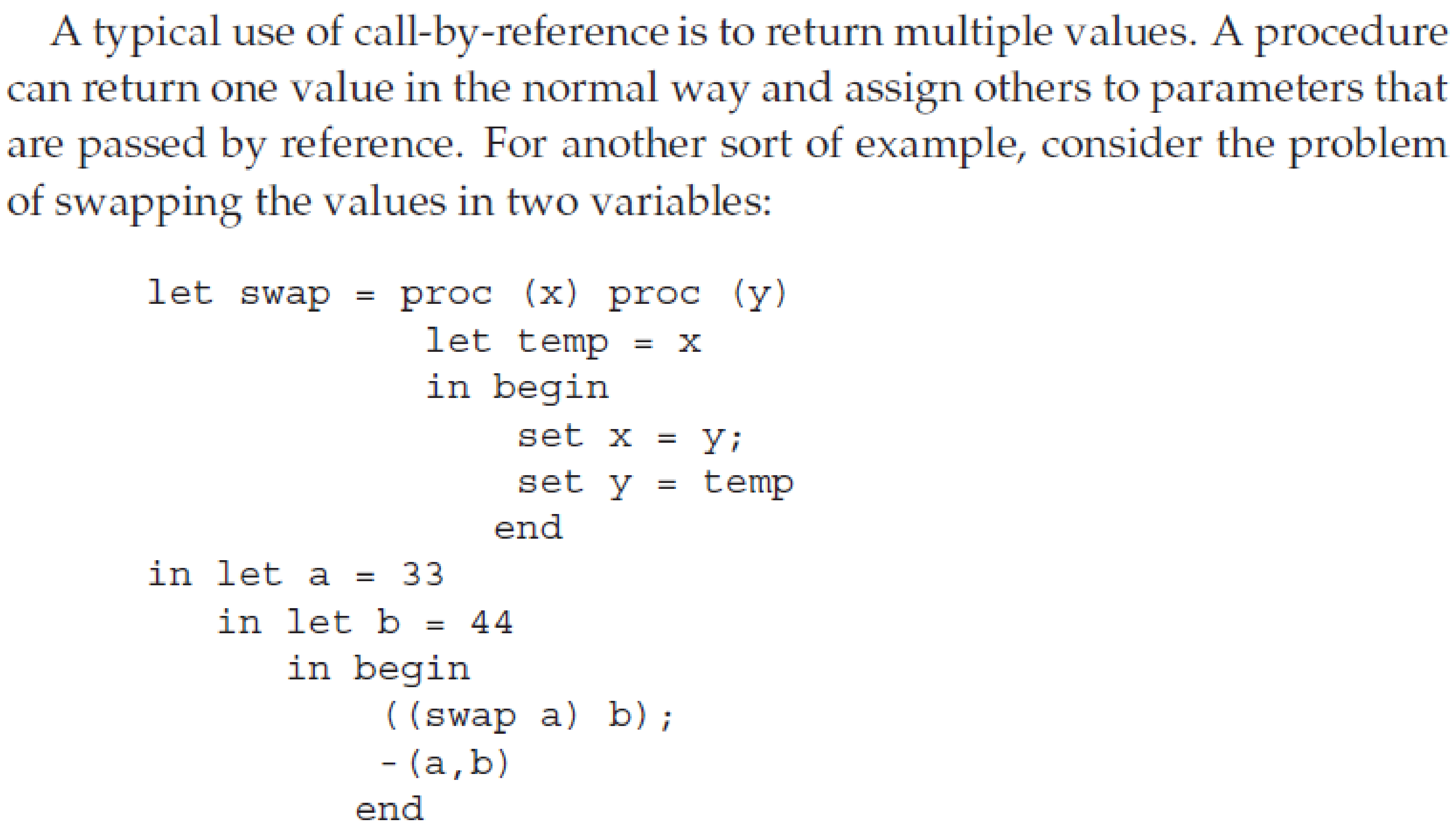


When a call-by-reference procedure is called and the actual parameter is a variable, what is passed is the *location* of that variable, rather than the contents of that location, as in call-by-value. For example, consider

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

When the procedure `g` is called, `y` is bound to the location of `z`, not the contents of that location. Similarly, when `f` is called, `x` becomes bound to that same location. So `x`, `y`, and `z` will all be bound to the same location, and the effect of the `set x = 44` is to set that location to 44. Hence the value of the entire expression is 44.

…



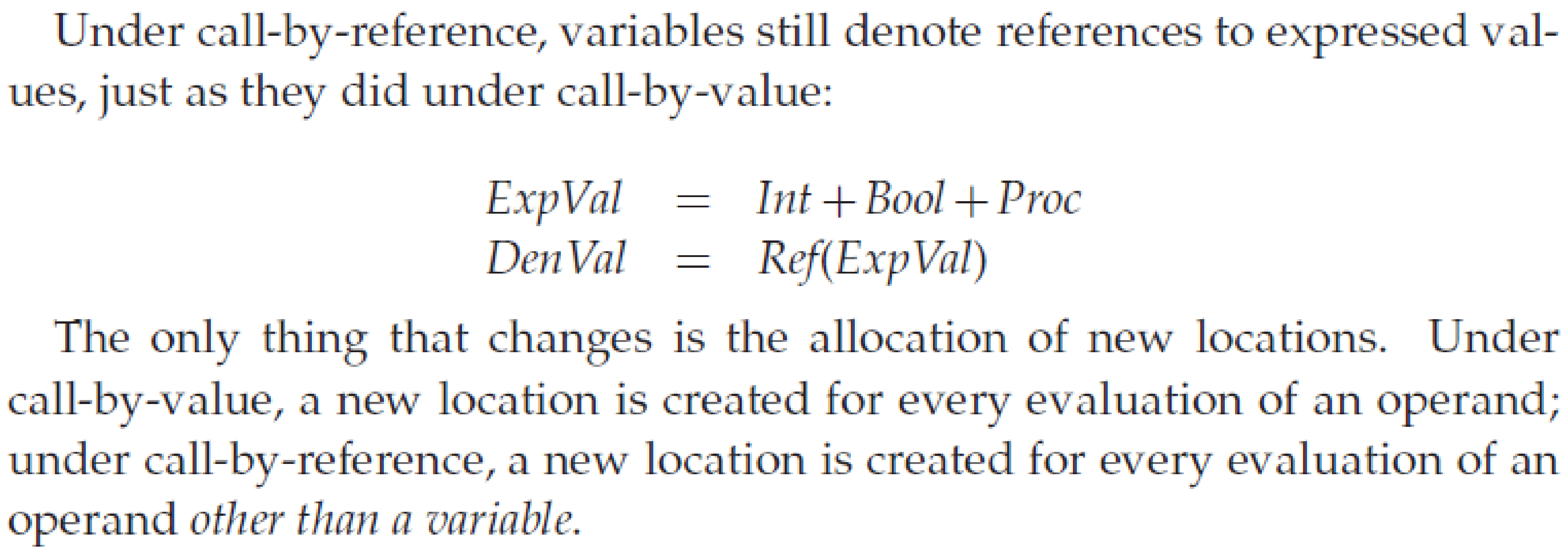
A typical use of call-by-reference is to return multiple values. A procedure can return one value in the normal way and assign others to parameters that are passed by reference. For another sort of example, consider the problem of swapping the values in two variables:

```
let swap = proc (x) proc (y)
            let temp = x
            in begin
                set x = y;
                set y = temp
               end
in let a = 33
   in let b = 44
      in begin
          ((swap a) b);
          -(a,b)
         end
```

…

Under call-by-reference, this swaps the values of `a` and `b`, so it returns 11. If this program were run with our existing call-by-value interpreter, however, it would return -11, because the assignments inside the swap procedure then have no effect on variables `a` and `b`.

CALL-BY-REFERENCE

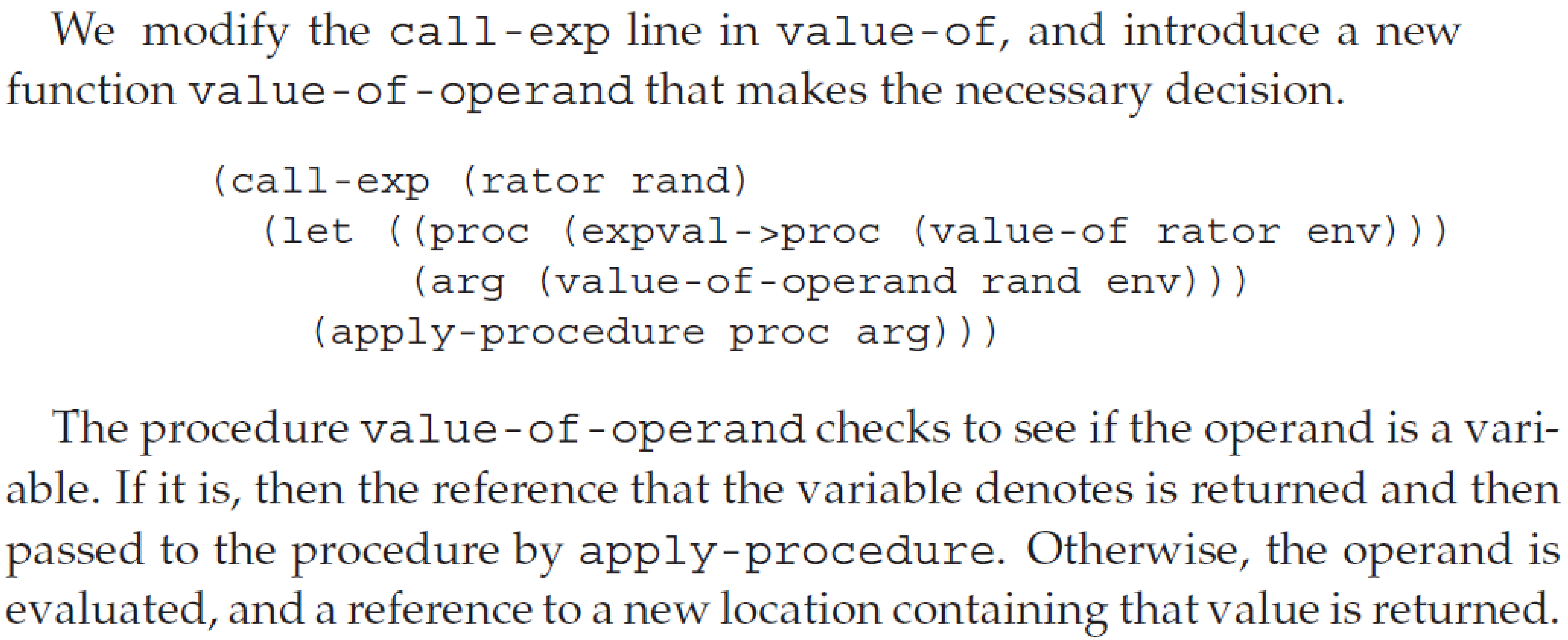


Under call-by-reference, variables still denote references to expressed values, just as they did under call-by-value:

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} \\ \mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal}) \end{aligned}$$

The only thing that changes is the allocation of new locations. Under call-by-value, a new location is created for every evaluation of an operand; under call-by-reference, a new location is created for every evaluation of an operand *other than a variable*.

CALL-BY-REFERENCE: Implementation

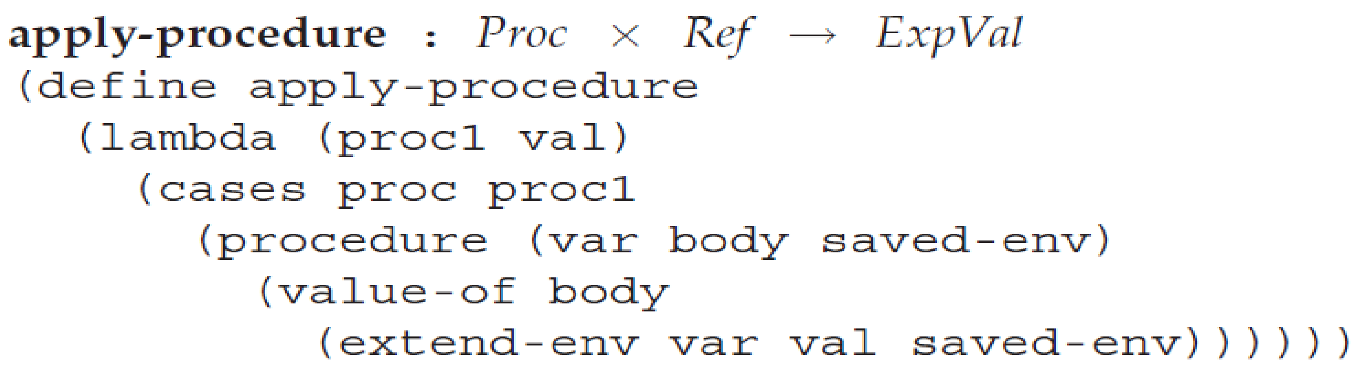


We modify the `call-exp` line in `value-of`, and introduce a new function `value-of-operand` that makes the necessary decision.

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of-operand rand env)))
    (apply-procedure proc arg)))
```

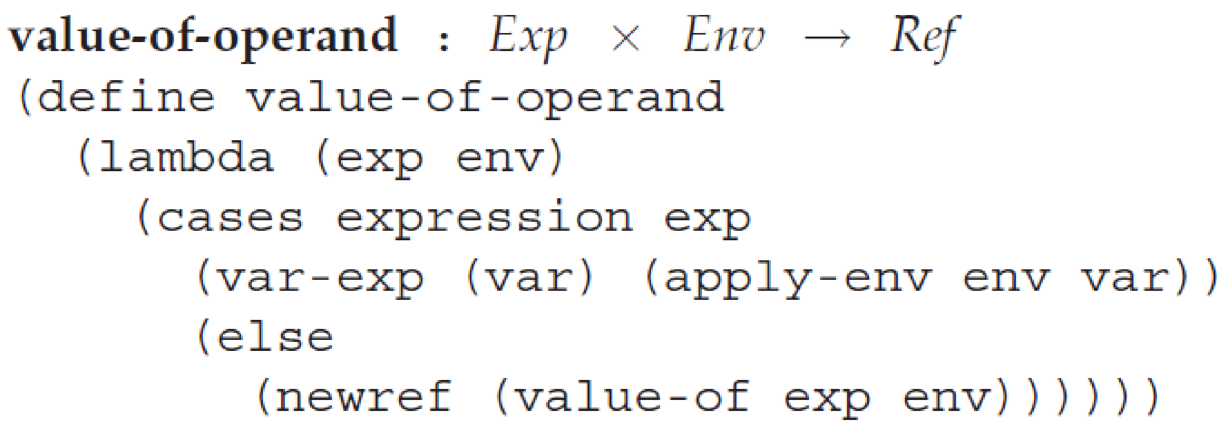
The procedure `value-of-operand` checks to see if the operand is a variable. If it is, then the reference that the variable denotes is returned and then passed to the procedure by `apply-procedure`. Otherwise, the operand is evaluated, and a reference to a new location containing that value is returned.

CALL-BY-REFERENCE: Implementation



apply-procedure : $Proc \times Ref \to ExpVal$

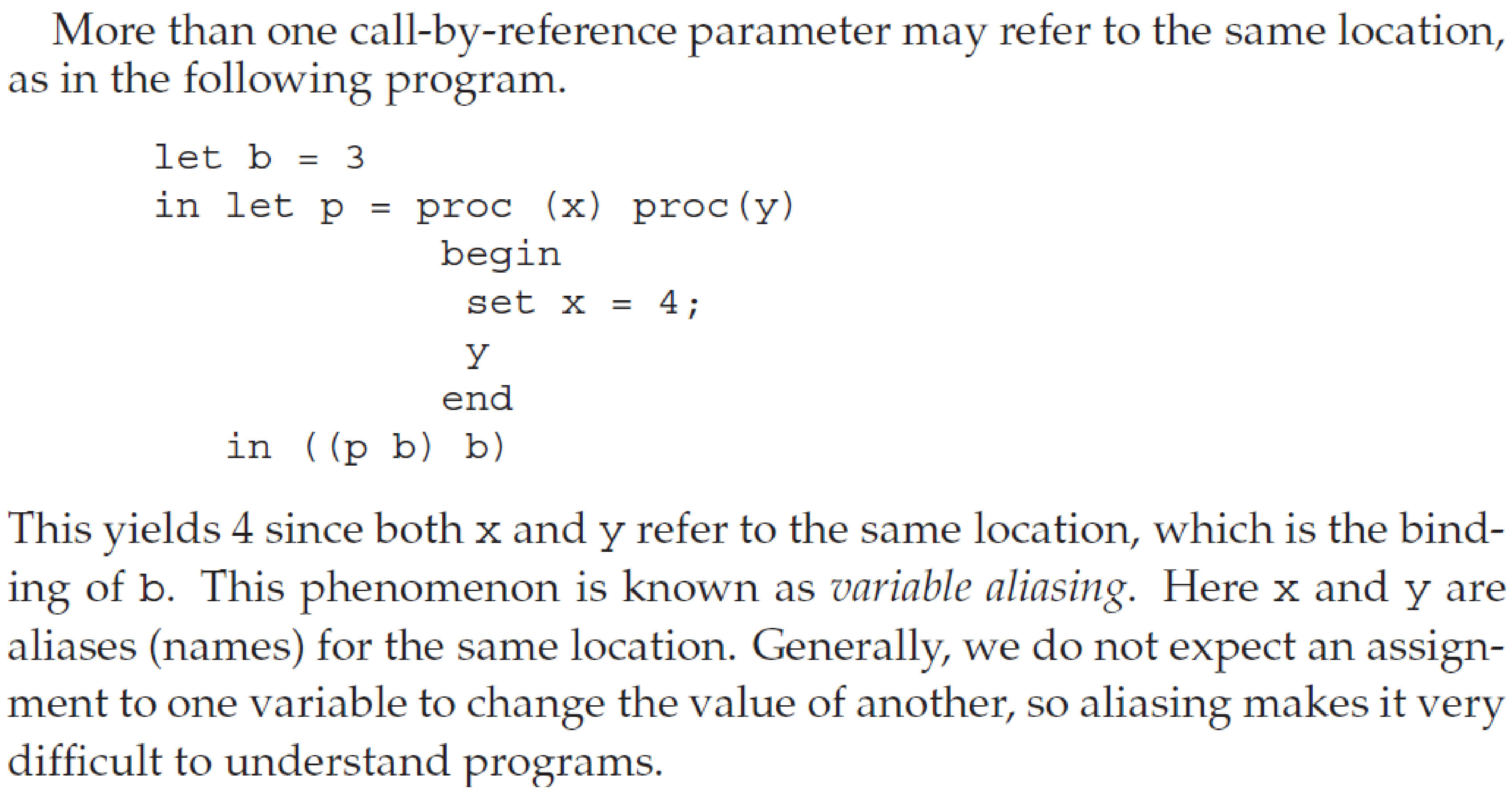
```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var val saved-env))))))
```



value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

CALL-BY-REFERENCE: Aliases



More than one call-by-reference parameter may refer to the same location, as in the following program.

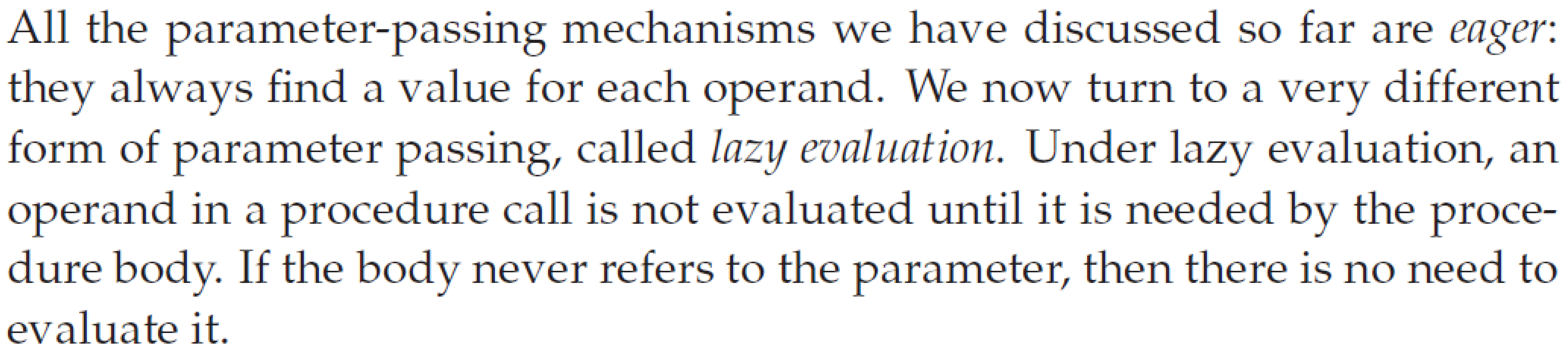
```
let b = 3
in let p = proc (x) proc(y)
            begin
             set x = 4;
             y
            end
   in ((p b) b)
```

This yields 4 since both x and y refer to the same location, which is the binding of b. This phenomenon is known as *variable aliasing*. Here x and y are aliases (names) for the same location. Generally, we do not expect an assignment to one variable to change the value of another, so aliasing makes it very difficult to understand programs.

Figure 4.14 Sample evaluation in CALL-BY-REFERENCE

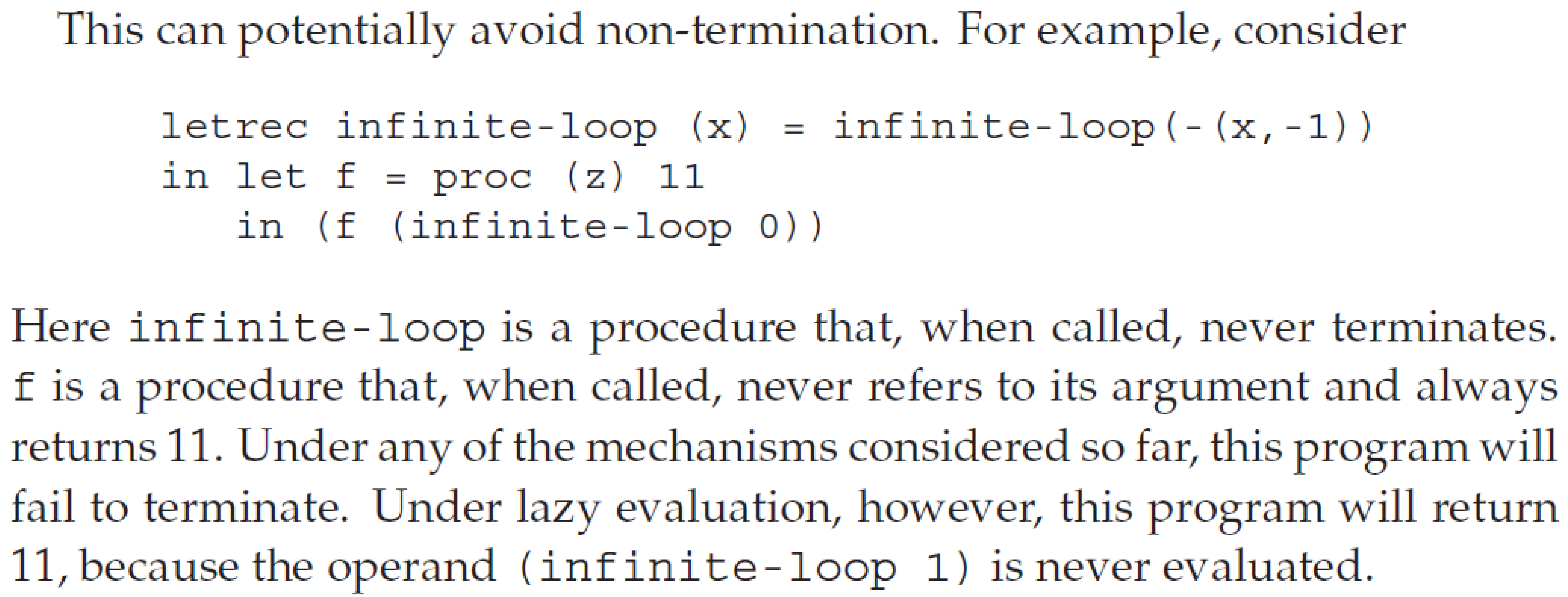
```
> (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
     (g z);
     z
  end")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))
entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

 entering body of proc y with env =
 ((y 5) (f 3) (i 0) (v 1) (x 2))
 store =
 ((0 #(struct:num-val 1))
  (1 #(struct:num-val 5))
  (2 #(struct:num-val 10))
  (3 (procedure x ... ((i 0) (v 1) (x 2))))
  (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
  (5 #(struct:num-val 55)))

 entering body of proc x with env =
 ((x 5) (i 0) (v 1) (x 2))
 store =
 ((0 #(struct:num-val 1))
  (1 #(struct:num-val 5))
  (2 #(struct:num-val 10))
  (3 (procedure x ... ((i 0) (v 1) (x 2))))
  (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
  (5 #(struct:num-val 55)))

 #(struct:num-val 44)
 >
```

Lazy Evaluation: CALL-BY-NAME and CALL-BY-NEED



All the parameter-passing mechanisms we have discussed so far are *eager*: they always find a value for each operand. We now turn to a very different form of parameter passing, called *lazy evaluation*. Under lazy evaluation, an operand in a procedure call is not evaluated until it is needed by the procedure body. If the body never refers to the parameter, then there is no need to evaluate it.

Lazy Evaluation: CALL-BY-NAME and CALL-BY-NEED

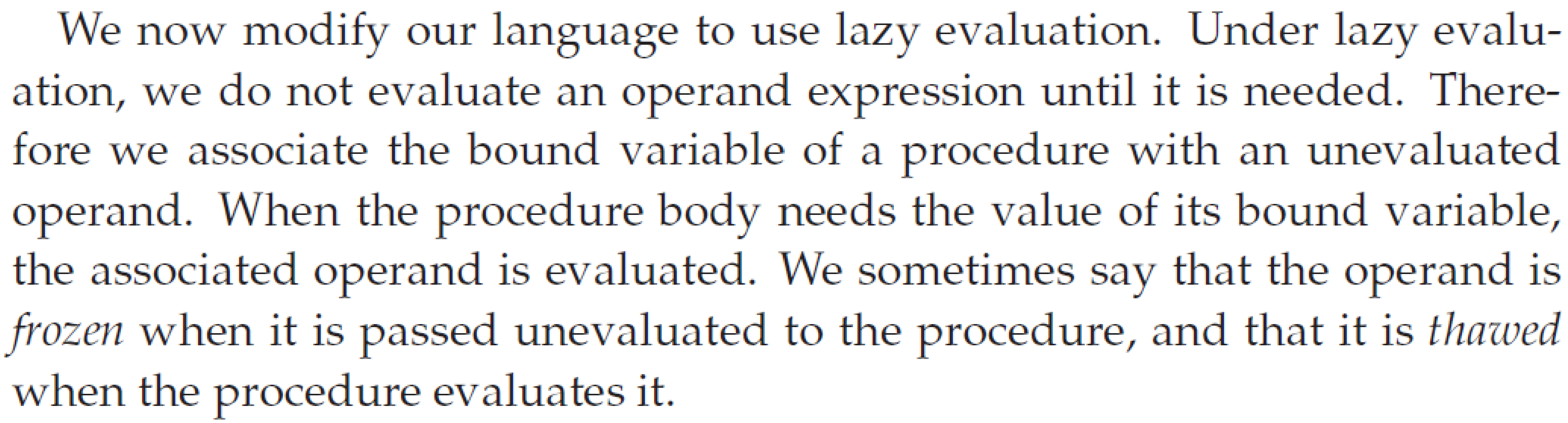


This can potentially avoid non-termination. For example, consider

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

Here `infinite-loop` is a procedure that, when called, never terminates. `f` is a procedure that, when called, never refers to its argument and always returns 11. Under any of the mechanisms considered so far, this program will fail to terminate. Under lazy evaluation, however, this program will return 11, because the operand `(infinite-loop 1)` is never evaluated.

...



We now modify our language to use lazy evaluation. Under lazy evaluation, we do not evaluate an operand expression until it is needed. Therefore we associate the bound variable of a procedure with an unevaluated operand. When the procedure body needs the value of its bound variable, the associated operand is evaluated. We sometimes say that the operand is *frozen* when it is passed unevaluated to the procedure, and that it is *thawed* when the procedure evaluates it.

…

Of course we will also have to include the environment in which that procedure is to be evaluated. To do this, we introduce a new data type of *thunks*. A thunk consists of an expression and an environment.

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

When a procedure needs to use the value of its bound variable, it will evaluate the associated thunk.

…

We therefore let our denoted values be references to locations containing either expressed values or thunks.

$$\begin{aligned} DenVal &= Ref(ExpVal + Thunk) \\ ExpVal &= Int + Bool + Proc \end{aligned}$$

Our policy for allocating new locations will be similar to the one we used for call-by-reference: If the operand is a variable, then we pass its denotation, which is a reference. Otherwise, we pass a reference to a new location containing a thunk for the unevaluated argument.

…

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

When we evaluate a `var-exp`, we first find the location to which the variable is bound. If the location contains an expressed value, then that value is returned as the value of the `var-exp`. If it instead contains a thunk, then the thunk is evaluated, and that value is returned. This design is called *call by name*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (value-of-thunk w)))))
```

The procedure `value-of-thunk` is defined as

value-of-thunk : *Thunk* $\rightarrow$ *ExpVal*
```
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

…

Alternatively, once we find the value of the thunk, we can install that expressed value in the same location, so that the thunk will not be evaluated again. This arrangement is called *call by need*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (let ((val1 (value-of-thunk w)))
          (begin
            (setref! ref1 val1)
            val1))))))
```